KIS Developer's Guide

Kabira Infrastructure Server

KIS Developer's Guide : Kabira Infrastructure Server

Published 27 Sep 2010

This document is intended to address *your* needs. If you have any suggestions for how it might be improved, or if you have problems with this or any other Kabira documentation, please send e-mail to pubs@kabira.com. Thank you! Feedback about this document should include the reference KISDG-32.2 . Updated on 27 Sep 2010 .

Contents

List of Figures

Part One

Part One: Creating KIS Applications

1

Introduction

This chapter introduces tools and terminology used in modeling and building a Kabira Infrastructure Server (KIS) component. KIS components are built directly from object-oriented models without a traditional coding stage. This makes them quick to develop and easy to maintain.

In this chapter you are introduced to:

- modeling and building components
- the terminology

You should be familiar with object-oriented design concepts.

KIS components

A KIS component is an executable model that can be deployed and reused. When you build a component, everything needed for deployment on a KIS node and for reuse in another component is wrapped into a single file.

Components let you create modular applications. You can model a related set of types and services in a single component, then access those types and services from any number of components. In addition, you can wrap interfaces to non-KIS services—for example, applications implemented in Java or accessed through CORBA—into components, and reuse those components in other components.

Although each component is deployed separately, you don't need to think about deployment when modeling a component. Dependencies between components are specified at build time, and are stored with the dependent component.

Steps to build a component

To build a KIS component you will:

- model your application logic in IDLos
- create a component specification describing how to build the model
- build the component using the Design Center server

Modeling a component Using a text editor, you model using the KIS interface definition language, IDLos. IDLos is a superset of IDL with entities and relationships added, and is UML compliant. You can think of it as a textual version of UML.

Component specification The description of how to build an KIS component is called a component specification. These component specifications are written in a text editor using a simple build specification language.

When specifying how you want the component to be built, you can import other components upon which your new component depends. You need to import any component that defines types or operations used in the component you are building.

Building a component Once your model and component specification are complete, you can build the component. The component specifications are sent to a KIS Design Center server to generate the deployable and reusable component.

The components are built on the platform where the Design Center server is running. You can model, specify, and build a component while working on one platform that will be deployed on another, simply by connecting to a Design Center server that is running on the deployment platform.

UML and IDLos

Throughout these books, models are expressed using both UML and IDLos. Figure 1.1 summarizes some basic elements of a UML class diagram

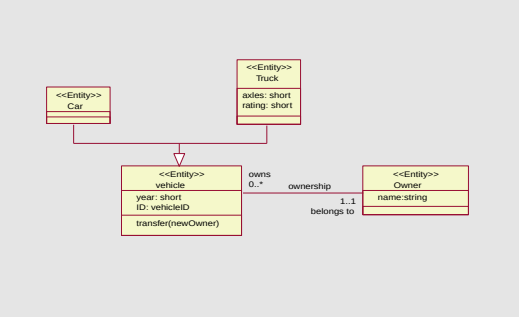

Figure 1.1. Basic features of the UML

Figure 1.1 shows four entities:

- zero or more Vehicles are owned by exactly one Owner. In IDLos this relationship would be represented as:

```
relationship ownership
{
    role Owner owns 0..* Vehicle;
};
```

- Vehicle has two attributes and an operation. The vehicle entity in IDLos would appear:

```
entity Vehicle
{
    attribute short year;
    attribute VehicleID ID;
    void transfer(in Owner newOwner);
};
```

- Car and Truck are two kinds of Vehicle, and Truck has two additional attributes beyond those inherited from Vehicle.

There are many good books on UML; this manual does not attempt to teach the notation. The UML chapter in Advanced KIS Modeling describes in detail how IDLos is represented in UML.

Terminology

Object-oriented analysis and design uses a number of different terms to represent similar concepts, varying from one methodology to another. KIS draws on a number of different traditions, so it is necessary to clarify the terminology that we use.

Term	Meaning
Attribute	Is a piece of data defined as part of an entity. Some traditions refer to this as a data member.
Component	The Design Center builds your application into one or more components that can be deployed on a KIS node.
Engine	A single executable process. You use a deployment specification to allocate components to the engines that run on a KIS node.
Entity	A model element that can be instantiated in the runtime as an object. Corresponds to a class in some methodologies.
Event	A data structure used to communicate information between objects. Events may be synchronous or asynchronous.
Interface	Indicates what part of an entity is exposed outside of a package. An entity may have many interfaces, but each interface may have only one entity. Other traditions have a broader definition.
Node	A managed area of shared memory that contains object data, the Event Distribution Bus, and other data structures.
Object	An instance of an entity.
Operation	Is an executable part of an entity, known in other traditions as a member function or as a method.
Signal	A message that causes a state transition in the state machine of an object. Known in some traditions as an event.

Term	Meaning
Supertype/subtype	Represents inheritance (also called generalization). If some entity B inherits from another entity A, then A is the supertype of B and B is a subtype of A. Other traditions may refer to this as parent class and child class, or as base class and derived class.

2

Creating KIS models

This chapter describes how you model in Kabira Infrastructure Server (KIS). It introduces each KIS modeling concept and shows how you use it to create KIS models. It provides you with modeling constructs, such as entities, attributes, operations, packages, that you can use in textual form to create your models.

The Unified Modeling Language is used to represent KIS models visually. IDLos, an IDL-based modeling language, provides the textual equivalent. This chapter introduces these modeling concepts. For more detailed reference material, refer to the following:

Chapter 3 explains the lexical rules of KIS models

Chapter 8 describes the KIS type system

Chapter 9 provides a detailed IDLos language reference

This chapter presents a general overview of models and then introduces the KIS modeling constructs:

- the section called " Relationship" on page 23
- the section called " State machine" on page 38
- the section called "Package" on page 30
- the section called "Interface" on page 33
- the section called "Local Interface" on page 37

The next sections explain how to model finite state machines, with a separate section covering each construct:

- the section called " State machine" on page 38
- the section called " State" on page 40
- the section called " Signal" on page 41
- the section called " Transition" on page 41

The final sections of this chapter cover:

- the section called "Inheritance" on page 42
- the section called " Namespaces" on page 50
- the section called "A big example" on page 53

Models

KIS applications are defined in terms of object models. Models are textually created in the IDLos modeling language using an ordinary text editor.

Consider a UML model fragment, as shown in Figure 2.1, and its corresponding IDLos representation.

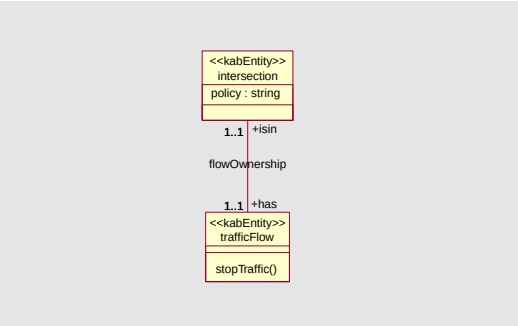

Figure 2.1. A UML model fragment

```
entity intersection
{
    attribute string policy;
};

entity trafficFlow
```

```
{
    void stopTraffic();
};

relationship flowOwnership
{
    role intersection has 1..1 trafficFlow;
    role trafficFlow isin 1..1 intersection;
};
```

This model fragment has two entities that are associated using a relationship. One entity has an attribute of type string, and the other has an operation that does not return a value. You would specify the implementation of this operation in Action Language using the action statement (not shown in this fragment).

Once created, your model is compiled into an application that you can deploy on a KIS server.

The KIS textual modeling language IDLos is based on IDL v 2.2, and existing IDL can be used directly in KIS applications. IDLos extends the standard, providing relationships, state machines, and triggers; as well as entities and actions to specify implementations.

Object oriented

The KIS modeling language is fully object-oriented (OO), supporting inheritance and polymorphism. It also supports some features often requested but not often found in OO languages: singletons, keys, triggers, and transactions.

In KIS, inheritance includes relationships, state machines, and triggers. The inheritance of relationships is particularly powerful.

Component oriented

You organize your KIS applications in packages. Objects in the same package have access to one another. To make objects visible outside a package, you define interfaces.

KIS packages are more than an analysis principle. Packages represent the lowest level of granularity for building components. You can build one component for each package or group several packages into one component. And because of the KIS runtime technology, you can add new components to your running application or update existing components without bringing down your application.

Richly typed

IDLos uses the IDL type system, which provides a rich system of basic and user-defined types. See Chapter 8 for more on the KIS type system.

Adaptable

You can use a range of KIS adapter factories to expose your model to any number of technologies, for example, CORBA, Java, PHP, and SNMP.

When you model with KIS, you ignore the specifics of the technology that you expose your model to. You make these implementation decisions later, at build time. While modeling, you concentrate solely on your business logic and the interfaces that expose your model.

An example

This example shows two packages; one is a client and the other is a server. The client will ask the server to say hello, and the server will respond by printing “Hello World”.

UML model The server consists of the entity TalkerImpl and the interface Talker that exposes the operation sayHi. The sayHi operation is implemented in the entity TalkerImpl. Clients will invoke the sayHi operation through the Talker interface.

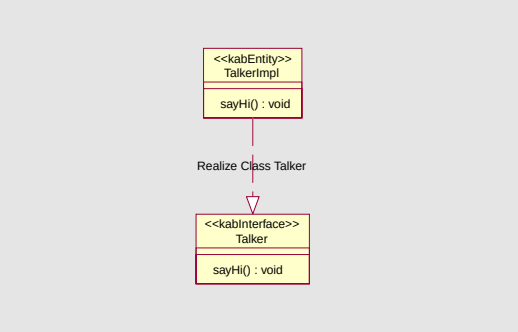

Figure 2.2. The server package

The operation sayHi is implemented with the following line of action language:

```
printf("Hello World\n");
```

The arrow connecting an interface with the entity that implements it may be drawn in either direction. The convention in this manual is to show the arrow pointing at the implementing entity, in the sense of the IDLos verb “expose”. You may wish to draw the arrow pointing to the interface, in the sense of the UML verb “realizes”.

The client package contains one local entity, Startup, with a lifecycle operation init that kicks off the application when the runtime initializes the client component.

Figure 2.3. The client package

The operation init contains the following lines of action language:

```
declare ::Server::Talker t;
declare ::swbuiltin::EngineServices es;
create t;
```

```
t.sayHi();
delete t;
es.stop(0);
```

IDLos Here is the same model in IDLos:

```
package Server
{
    entity TalkerImpl
    {
        oneway void sayHi();
    };
    interface Talker
    {
        oneway void sayHi();
    };
    expose entity TalkerImpl with interface Talker;
};
action ::Server::TalkerImpl::sayHi
{‘
    printf(“Hello World\n”);
‘};

package Client
{
    [local]
    entity Startup
    {
        [initialize]
        void init();
    };
};
action ::Client::Startup::init
{‘
    declare ::Server::Talker t;
    declare ::swbuiltin::EngineServices es;
    create t;
    t.sayHi();
    delete t;
    es.stop(0);
 ‘};
```

Entity

Modeling with KIS means applying an object-oriented approach. Central questions of any object-oriented approach include

- what types of objects will my application be dealing with?
- what type of data will the objects provide?
- what behavior will the objects display?

For example, one type of object in your application could be specific customer records. The data of a customer record could include a customer’s name, address, and phone number. The behavior of a customer record could involve changing the customer data, for example, when a customer moves or marries.

In KIS to model a type (or class) of object, you use an entity. An object of that type is an instance of the entity. In an application there can be many instances of one entity.

An entity allows you to define the internal data structures (in the form of attributes) and behavior (as operations) for a type of object. For entities with many instances you can define keys to provide efficient selection of individual objects. Triggers let you invoke behavior when certain events happen, such as creating a new object or changing an attribute value. You can model the lifecycle, or states, of the entity's instances in a state machine.

Entities are the most powerful user-defined type. They are at the heart of every KIS model.

The following table shows the modeling elements that you use to define an entity's data and behavior:

Element	Description
the section called " Attribute" on page 15	Data member of an entity. Each instance has a data member of the type specified by an attribute.
the section called "Operation" on page 16	Behavior that you invoke on an instance.
the section called " Key" on page 22	Unique identifier. Keys enable you to efficiently query for individual instances.
the section called "Entity trigger" on page 25	An event-operation pair. A trigger specifies an operation that is invoked on an instance at the onset of a certain event, for example, when you create, delete, refresh, or relate an instance.
the section called " State machine" on page 38	Finite state machine. A state machine defines the life cycle of an instance, the states through which each instance can pass.

Aspects of entities

This section summarizes some of the aspects of an entity: nested types, namespace, relationships, inheritance, and exposure.

Namespace Each entity lives within some namespace (see the section called " Namespaces" on page 50).You can define an entity only within the direct scope of either a package or a module.

Relationships You can define a relationship between entities, so that instances of one entity can be linked to instances of the other entity. See the section called " Relationship" on page 23.

Inheritance An entity may inherit traits from another entity. See the section called "Inheritance" on page 42.

Exposure There is no access to an entity from outside its package unless you explicitly enable it. You enable access to an entity's attributes and operations from outside the package using an interface. See the section called "Interface" on page 33.

IDLos

Use the entity statement (see "entity" on page 359 for syntax details).

Here is the IDLos for the example above:

```
entity TimerEventImpl
{
    // attributes
    attribute Road road;
    // operations
    oneway void generate ();
};
```

Entity properties

You can tag an entity with properties that affect the number of permitted instances as well as its representation in shared memory.

The following table shows the properties you can use in the definition of an entity.

Property	Description
the section called “ singleton” on page 13	Only one instance of a singleton entity is ever instantiated.
the section called “extent-less” on page 13	The extent (a list of all instances) is not locked during object creation and deletion.
managedextent [???]	The extent (a list of all instances) is locked during object creation and deletion.
dynamiclog	The runtime allocates the entity's log when you modify an instance and frees the log when the transaction commits or aborts. This reduces the shared memory footprint of instances; use this property for instances which you modify infrequently.

Setting properties In IDLos entity properties appear at the beginning of the IDLos statement, in square brackets.

```
[singleton] entity PortAllocator {};
[extentless] entity InvoiceItem {};
```

singleton The singleton property denotes an entity that can have only one instance. At runtime, creating a singleton either:

- creates an instance if one does not already exist, otherwise it
- returns a reference to the existing instance

Singletons must not be local, and cannot be a supertype or subtype of another entity.

extentless You can use the extentless property to suppress the locking of extents in the application server.

KIS keeps the extent of an entity in shared memory. The extent is a list of all instances of the entity. Even when an instance is flushed from shared memory, a reference to it stays in the extent. The extent is normally locked when creating, deleting, and locating objects.

Extentless entities do not lock the extent. This places certain restrictions on what you can do, and it optimizes the behavior of the runtime under certain conditions.

When you iterate across all the instances of an extentless entity, you may miss some objects or encounter some twice, depending on other transactions' use of that type. Similarly, the result of the cardinality operator is not definitive for extentless entities.

So why would you use extentless objects? Because they may work better in your design for objects that are either:

- extremely numerous, and the extent itself consumes too much memory
- created and deleted frequently and concurrently, causing lock contention on the extent

This second point requires some explanation. When normal objects (as opposed to extentless ones) are created or deleted, their extent is locked until the completion of the transaction. This prevents other instances of the entity from being created or destroyed until the transaction completes.

When an extentless object is created or deleted, locking takes place at a much lower level and for a much shorter period. Once the object creation is complete, other instances of the type can be created right away. Using extentless objects can substantially improve performance in certain cases.

extentless is the default behavior for entity extents.

managedextent You can use the managedextent property to force locking of extents during object create and deletion.

KIS keeps the extent of an entity in shared memory. The extent is a list of all instances of the entity. Even when an instance is flushed from shared memory, a reference to it stays in the extent. Entities that use the managedextent property lock the extent when creating, deleting, and locating objects.

So why would you use managedextent objects? Because they may work better in your design for objects that are either:

- the cardinality of objects in the extent must be accurate
- iterating over the extent must produce predicatable and repeatable results

When managedextent objects are created or deleted, their extent is locked until the completion of the transaction. This prevents other instances of the entity from being created or destroyed until the transaction completes.

When an managedextent object is created or deleted, locking takes place at the transaction boundaries. The duration of this lock is much longer than the locking that takes place for extentless objects. This lock also causes serialization of creates and deletes for managedextent objects.

Local entity

Local entities are like regular entities, but with some limitations. Normally you will only use local entities for engine lifecycle operations, e.g. initialize, terminate, etc. operations. Local entities have a number of restrictions:

- not distributable
- not recoverable
- cannot be used by clients in other components

- cannot be used as references in distributable types
- cannot be used as parameters in non-local operations

Local entities must not be singletons.

Local entities do not have to be created or deleted. They can be used after they are declared without an explicit create. They are automatically destroyed when the code block in which they were declared goes out of scope.

Aspects of local entities

This section discusses the exposure and lifecycle operations of local entities.

Exposure There is no access to a local entity from outside its package unless you explicitly enable it. You enable access to a local entity's operations with the following element:

Element	Description
the section called "Local Interface" on page 37	Defines operations that are accessible from outside a local entity's package.

Lifecycle operations Use operations defined in local entities to perform lifecycle tasks. You specify that an operation is invoked when the runtime initializes, is reloaded, or terminates a component (see the section called " initialization, unload, reload, termination" on page 20) or before any other lifecycle operation (see the section called "packageinitialize, packagereload" on page 21).

IDLos

Use the entity statement. A local entity is an entity marked with the local property (see "local entity" on page 98 for syntax details).

```
[local]
entity StartUp
{
    // operations
    [initialize]
    void initData ();
};

[local]
entity Utilities
{
    void who();
    void whatis();
};
```

Attribute

Attributes specify the data members of entities. You define an attribute in an entity or an interface. Defining an attribute in an interface enables cross-package access to the same attribute in the exposed entity.

An attribute has a type and an attribute label. Attributes can be of any valid type.

You can define a trigger that invokes an operation when an attribute is accessed (see the section called “Attribute trigger” on page 27 for details).

IDLos

Use the attribute statement (see “attribute” on page 85 for syntax details).

Here is an IDLos example:

```
entity ModeRequest
{
    // attributes
    attribute ModeType modeType;
    attribute RoadDirection modeDir;
    attribute long timeGreenNS;
    attribute long timeYellowNS;
    attribute long timeGreenEW;
    attribute long timeYellowEW;
    attribute boolean timeChange;
};
```

Read-only attributes

You can optionally use the readonly keyword to specify that an attribute may not be modified. Attributes are read-write by default unless you specify readonly.

Specifying a read-only attribute In IDLos you use the readonly keyword at the beginning of the attribute statement:

```
readonly attribute long employeeIndex;
```

Dirty read attributes

You can optionally define an attribute to be read without taking a transaction lock. This is called dirty reads. If an attribute is defined to be a dirty read mulitlpe reads of the attribute in the same transaction are not guarantted to return the same value since there is no transaction lock taken on the attribute when it is read.

Specifying a dirtyread attribute In IDLos you use the dirtyread property on the attribute in both the interface and entity definition of the attribute.

```
[ dirtyread ] attribute long employeeIndex;
```

Operation

Operations provide a way to invoke actions on an object. You define an operation in an entity or an interface. Defining an operation in an entity provides an entry point for invoking behavior; defining it in an interface exposes that operation to other packages.

Operations can be invoked and executed at any time. In contrast, use a state machine to specify behavior that takes place as objects pass through different states.

Aspects of operations

Each operation has a signature: name, parameters, and return type. An operation may also raise exceptions. An operation's name must be unique within the entity's namespace, and cannot be overloaded.

Parameters Operations can take parameters. Parameters have a type and a direction: in, inout, and out. Parameters with the in direction cannot be changed by the operation's implementation. Parameters with the inout and out direction may be changed.

You cannot call a non-const operation on a const reference. All in parameters are passed in as const references. Remember that in a const operation, self is a const reference.

Return type The return type can be any type valid in the context of the operation, or void if there is no type returned.

Exceptions You must specify the exceptions that an operation raises (for more on exceptions, see the section called " exception" on page 179).

You implement operations using action language (for more on action language, see Chapter 7).

One-way and two-way operations By default, operations are two-way, or synchronous, operations. The work is dispatched to the destination and the caller blocks (suspends execution) until the operation returns. But you can explicitly define a one-way, or asynchronous, operation. When a caller invokes a one-way operation, the caller keeps executing even though the work may not yet be completed.

IDLos

For a detailed description of the IDLos syntax for operations, see "operation" on page 103.

Here is an IDLos example:

```
entity GUIProxyImpl
{
    // operations
    void updateMode (
        in ModeType modeType,
        in RoadDirection modeDir);
    void updateSensor (
        in boolean newCar,
        in SideDirection carDir);
    LightColor getLightN () raises (expLightNotFound);
    void addRequest (inout ModeRequest req);
    void getAllLights (
        out LightColor north,
        out LightColor south,
        out LightColor west,
        out LightColor east);
};
```

Operation properties

The following table shows the properties you can use in the definition of an operation.

Property	Description
the section called "local" on page 19	The runtime invokes a local operation directly, rather than dispatching it through the event bus (see the section called "local" on page 19).
the section called "const" on page 19	The operation cannot change the internal state of the object (see the section called "const" on page 19).
	The operation is asynchronous; all other operations are twoway, or synchronous
the section called "virtual" on page 19	Defines a polymorphic operation (see the section called "virtual" on page 19).
initialize	Lifecycle property (applies only to operations defined in a local entity): The runtime invokes the operation when the operation's component initializes (see the section called " initialization, unload, reload, termination" on page 20).
inline	The operation is optimized for calling locally. inline operations cannot be exposed through an interface, cannot be virtual, and can only be called from the entity in which they are defined.
packageinitialize	Lifecycle property (applies only to operations defined in a local entity. The runtime invokes the operation when an engine is being started before any work is dispatched. This operation can only access entities in the same package in which it is implemented.
packagereload	Lifecycle property (applies only to operations defined in a local entity. The runtime invokes the operation after an engine has been reloaded before any work is dispatched. This operation can only access entities in the same package in which it is implemented.
pure	Specifies that a virtual operation has no implementation in the base type; the implementation must be provided in the derived types. (May only be used together with the virtual property.)
reload	Lifecycle property (applies only to operations defined in a local entity. The runtime invokes the operation when an engine has been reloaded.
terminate	Lifecycle property (applies only to operations defined in a local entity): The runtime invokes the operation when the operation's component terminates (see the section called " initialization, unload, reload, termination" on page 20).

Property	Description
the section called "packageinitialize, packagereload" on page 21	Lifecycle property (applies only to operations defined in a local entity): The runtime invokes the operation before all other lifecycle operations when the operation's component initializes (see the section called "packageinitialize, packagereload" on page 21).
lockmode	Used with the value "dirtyread" to enable dirty reads of the attribute. The default mode is "normal." With dirty read enabled, KIS will not set a read lock when reading the attribute.
unload	Lifecycle property (applies only to operations defined in a local entity. The runtime invokes the operation before an engine is reloaded.

Setting operation properties In IDLos operation properties appear at the beginning of the IDLos statement, in square brackets:

```
[oneway, virtual]
void addRequest (in ModeRequest req);
```

local When you build a component that invokes a local operation, you must link in the local operation's implementation.

Local entities may be used as operation parameters to local operations, or as parameters to operations defined in a local entity (see the section called " Local entity" on page 14 for more).

const The const property restricts what an operation can do. The restrictions that it imposes are:

- a const operation cannot change the internal state of the object
- the action language that implements the operation may not invoke any non- const operations

If an operation in an entity is marked const, then it must also be marked const in any interfaces that expose it. Similarly, if it is not marked const, then it must not be marked const in the interface. This can be stated as:

An exposing operation in an interface must match the "constness" of the operation in the underlying entity.

virtual This property forces an operation to be dispatched polymorphically. This means operations are invoked based on the actual instance type, rather than on the type of the handle. So even when you upcast a subtype object to the supertype, if an operation is marked virtual in the supertype, the subtype operation is invoked. See the section called " Virtual operations and polymorphic dispatch" on page 46.

inline This property causes the model complier to generate optimized code for dispatching this operation in the same entity in which it was declared. inline operations have the following limitations:

- can only be called from the entity in which it is declared
- cannot be declared virtual

- cannot be exposed through an interface

pure This property may only be used together with the virtual property, typically in the form [pure, virtual]. It indicates that there is no supertype implementation, and that derived types must provide an implementation.

lockmode KIS provides a "dirty read" mode to specify attributes that do not always require read locks. You enable this mode using the property "lockmode=dirtyread" on the attribute in the entity definition:

```
entity thingImpl
{
    [ lockmode=dirtyread ]
    attribute long myAttr;
};
```

This allows you to then define interfaces that use dirty reads. To define the dirty read in an interface:

```
interface DirtyReadThing
{
    [ lockmode=dirtyread ]
    attribute readonly long myAttr;
};
expose entity thingImpl with interface DirtyReadThing;
```

You can also define "clean" interfaces that do set read locks:

```
interface CleanReadThing
{
    [ lockmode=normal ]
    attribute readonly long myAttr;
};
expose entity thingImpl with interface CleanReadThing;
```

Accesses directly to the entity attribute (not via an interface) always use read locks.

Omitting the lockmode property is equivalent to specifying "lockmode=normal."

initialization, unload, reload, termination You can mark operations so that the KIS runtime invokes them upon component initialization, reload, or termination. You may define any number of such operations in any number of local entities.

An initialization operation is called when an engine starts. A terminate operation is called when the engine exists. An unload operation is called before an engine is reloaded. The reload operation is called after the reload completes.

You do not know the order that these three types of lifecycle operations are invoked. For example, if you define multiple initialize operations in a component, then these operations may execute in any order (although operations in singletons will execute before operations in other entities).

Although the initialize and reload operations are invoked when the engine starts, it is possible that external events (from other engines) may be processed before the lifecycle operation. Where this could cause a problem, use the packageinitialize or packagerealod property described below.

packageinitialize, packagereload You can use the packageinitialize or packagereload property, to designate an operation that executes before any other lifecycle operation. It also executes before any external events are processed; this lets you clean up process resources on initialization and reload.

The packageinitialize operation is called when an engine is started. The packagereload operation is called after an engine is reloaded.

Lifecycle example The example contains a local entity with several lifecycle operations. The runtime invokes init and initializeServices when the component is initialized, intializeServices and checkNetConnections when it recovers the component, and cleanUp when it terminates the component.

Consider the local entity EngineEvents in the following figure.

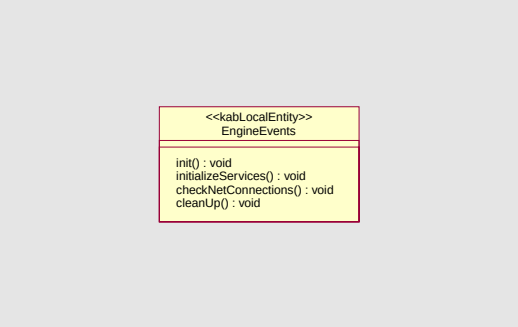

Figure 2.4. Lifecycle operations

IDLos Here is IDLos for the example above:

```
[local]
entity EngineEvents
{
    [initialize]
    void init();

    [
        initialize
    ]
    void initializeServices();

    [terminate]
    void cleanUp();
};
```

Automatic target instance All these lifecycle operations implicitly create a new instance of the entity and invoke the operation on that instance; the instance is destroyed when the operation completes. (If the entity is a singleton, the instance is created only if it does not already exist, and is not deleted when the operation completes.)

Although you can have lifecycle operations in any entity, it is better to use local entities only. The auditor will warn you if you use lifecycle operations on non-local entities.

Key

A key allows is used to select one or more instances of an entity (see the section called "Entity" on page 11). You define a key in an entity, interface, or a relationship (see the section called "Interface" on page 33). Defining a key in an interface enables cross-package access to the key in the exposed entity. Keys in a relationship contain only the instances in the relationship, instead of the entire extent.

You define a key to use one or more attributes (see the section called " Attribute" on page 15). Each key has a name.

When you construct a query in action language, and you use all the attributes of the key in your where clause (see "select" on page 436), KIS optimizes the search.

If a key appears in an interface, the identical key must exist in the entity that implements that interface. If an interface exposes all the attributes used in a key, but does not expose the key, the Design Center auditor will generate a warning.

You may not define a key in an entity that has any operations tagged with a lifecycle property (see the section called "Lifecycle operations" on page 15).

There is currently no key support for attributes of these types: sequence, or array.

Unique and Non-Unique

Keys can be unique or non-unique.

A unique key enforces that only one instance with the attribute value(s) specified in the key exist. If another instance is created with the same attribute values a non-unique key exception is thrown. Unique keys are accessed using the action language select or for statement.

A non-unique key allows multiple instances with the same attribute value(s) in the key to exist. Non-unique keys are accessed using the action language for statement.

The cardinality of a key is specified using the unique property. Setting unique = true, defines the key as unique, unique = false, defines the key as non-unique. The default value of the unique property is true.

Ordered and Non-Ordered

Keys can be ordered or non-ordered.

An ordered key allows action language for statements to specify ascending or descending order. Ordering can be specified on both unique and non-unique keys. The order of duplicate instances for the same key value in a non-unique key is undefined.

The ordering of a key is specified using the ordering property. Setting ordered = true, defines the key as an ordered key, ordered = false, defines the key as non-ordered. The default value of the ordered property is false.

IDLos

Use the key statement (see “key” on page 95 for syntax details). List the attributes comprising a key as a comma-separated list enclosed in braces.

```
entity employee
{
    attribute string firstname;
    attribute string middleinitial;
    attribute string lastName;
    attribute long SSN;
    attribute long employeeNumber;

    [ unique = true, ordered = false ]
    key first {SSN};

    [ unique = true, ordered = true ]
    key second {employeeNumber, SSN};
};
```

Relationship

Relationships define associations between entities. They let you link two instances, using relate in action language, or unlink them, using unrelate. When instances are linked, you can use action language to navigate across the links, to retrieve an associated instance, or retrieve a set. “Handling relationships” on page 129 has more information on relate, unrelate and navigation.

Relationships are a powerful abstraction, relieving you from writing numerous accessors and handling relationship integrity. The KIS server optimizes link storage, set retrieval, and keyed searches. All the relates and unrelates are transactional. Relationship integrity is part of KIS server transactions.

Relationship navigability

You can navigate a relationship from one object to the next, or back the other way. A relationship in KIS has one or two roles defined to provide navigation in one or both directions.

Roles Roles have a name, a from entity, a to entity, and a multiplicity. A relationship containing just one role is a one-way relationship, and cannot be navigated from the other side of the role.

Relationships may freely cross package boundaries. The relationship, the from entity, and the to entity can even be defined in three different packages.

There is no guarantee that a relationship will be selected in the same order in which it was related. Ordering can be maintained with additional reflexive relations such as linked lists or using ordered keys.

Example

The following UML diagram represents a relationship between two entities (as solid lines). The name of the relationship appears next it. Role names and multiplicity appears next to the to entity.

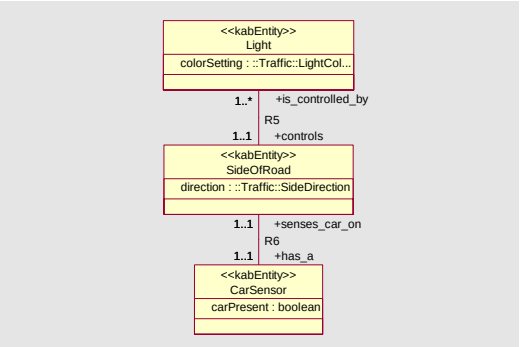

Figure 2.5. Relationship example

This example in the figure above defines two relationships. The textual descriptions are:

- Relationship R5: every SideOfRoad is controlled by one-or-more Lights, and each Light controls one-and-only-one SideOfRoad.
- Relationship R6: every CarSensor senses a car on one-and-only-one SideOfRoad, and every SideOfRoad has one-and-only-one CarSensor.

The example shows a role named controls. Light is the from entity; SideOfRoad is the to entity. And the multiplicity is 1..1.

You can also specify a relate and unrelate trigger for each role (see the section called “Role trigger” on page 28 for more on role triggers).

IDLos

Use the relationship statement (for syntax details, see “relationship / role” on page 377).

Here is the IDLos for the example in Figure 2.5:

```
relationship R5
{
     role Light controls 1..1 SideOfRoad;
     role SideOfRoad is_controlled_by 1..* Light;
};
relationship R6
{
     role SideOfRoad has_a 1..1 CarSensor;
     role CarSensor senses_car_on 1..1 SideOfRoad;
};
```

Associative relationships

Sometimes, there is data associated with each link between two instances. For example, a separate marriage certificate belongs to every marriage, containing the wedding date, the location, and officiating official.

Example The following example demonstrates an association between a relationship and an entity (as a dashed line), as shown in the figure below.

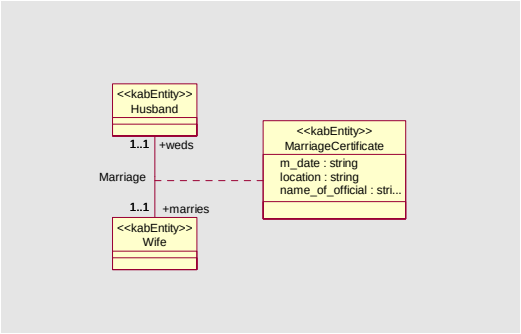

Figure 2.6. An associative relationship using an associative entity

IDLos Use the using statement to set up an association between a relationship and an entity.

```
entity Husband {};
entity Wife {};
entity MarriageCertificate
{
    string date;
    string location;
    string name_of_official;
};
relationship Marriage
{
    role Husband marries 1..1 Wife;
    role Wife weds 1..1 Husband;
    using MarriageCertificate;
};
```

The using phrase in the relationship block indicates which entity to use for the association.

Entity trigger

Entity triggers allow you to specify operations that the runtime invokes at the onset of certain types of events, for example, when your application instantiates an entity or deletes an instance. You define an entity trigger within the context of an entity.

Trigger types

When you define an entity trigger, you can only use operations that you define directly in the entity or in a supertype.

Trigger type	Invokes the trigger operation when...
commit	a transaction in the entity commits (see the section called “commit, abort” on page 26).
abort	a transaction in the entity aborts (see the section called “commit, abort” on page 26).
create	you create an instance of entity type (see the section called “create, refresh, state-conflict, update” on page 26).

Trigger type	Invokes the trigger operation when...
delete	you delete an instance of entity type (see the section called “create, refresh, state-conflict, update” on page 26).
refresh	you refresh an instance of entity type, from a remote node or data store (see the section called “create, refresh, state-conflict, update” on page 26).
state-conflict	a state conflict occurs.
update	you update an instance of an entity by setting one of its attributes. The update trigger is called when the entity handle goes out of scope.

commit, abort The trigger operation must be in the entity and satisfy the following conditions:

- it is two-way (cannot be a signal or oneway)
- its return type is void
- it has no parameters

Abort and commit triggers must be implemented by two-way operations, and need to be enabled in the action language in order for them to be called. e.g.:

```
action doit
{`
    // enable abort trigger, then do stuff that
    // may deadlock.
    declare ::swbuiltin::ObjectServices os;
    os.enableAbortTrigger(self);
    GoDoSomething();
`};
```

Abort and commit triggers must not take any locks—because there is no valid transaction context for the lock.. Doing so causes a run-time error with unpredictable results. Be very careful when writing actions that implement abort and commit triggers. Refer to the section on locks in Advanced KIS Modeling to understand what actions may take a lock.

create, refresh, state-conflict, update The trigger operation must be in the entity and satisfy the following conditions:

- it is two-way, oneway, or signal
- its return type is void
- it has no parameters

If you specify initial values in a create statement (see “Manipulating data” on page 110), then the create trigger fires after the values are set.

delete The trigger operation must be in the entity and satisfy the following conditions:

- it is two-way

- its return type is void
- it has no parameters

Create and delete triggers with inheritance: If you define create triggers for both the supertype (parent) and subtype (child), then the supertype trigger fires before the one in the subtype. Conversely, subtype delete triggers fire before those in a supertype.

IDLos

Use the trigger statement (for syntax details, see "trigger" on page 113).

The definition appears within the scope of the entity to which it applies.

The following IDLos shows several examples of entity triggers:

```
entity PedestrianLight
{
    // operations
    void powerup();
    void cleanup();
    void reset();
    void changed();

    // triggers
    trigger powerup upon create;
    trigger cleanup upon delete;
    trigger reset upon refresh;
    trigger chnaged upon update;
};
```

Attribute trigger

Attribute triggers allow you to specify operations that the runtime invokes at the onset of certain types of events, for example, when your application sets or accesses an attribute value.

Trigger types

You can define attribute triggers to occur both before or after an attribute is set, or before or after an attribute is retrieved.

Trigger type	Invokes the trigger operation when…
pre-get	before you access the value assigned to an attribute.
post-get	after you access the value assigned to an attribute.
pre-set	before you assign a value to an attribute.
post-set	after you assign a value to an attribute.

pre-get, post-get, pre-set, post-set The trigger must be defined in the same entity as the attribute to which it applies. The trigger operation can be inherited. The trigger operation must satisfy the following conditions:

- it is one-way, two-way, or a signal

- its return type is void
- it has no parameters

IDLos

Use the trigger ... upon statement (for syntax details, see the section called “ trigger” on page 193

The following IDLos shows examples of each type of attribute trigger:

```
entity Light
 {
     // attributes
     attribute LightColor colorSetting;

     // operations
     void precondition();
     oneway void postcondition();
     oneway void broadcast();

     // signals
     signal logAccess();

     // triggers
     trigger precondition upon pre-set colorSetting;
     trigger postcondition upon post-set colorSetting;
     trigger logAccess upon pre-get colorSetting;
     trigger broadcast upon post-get colorSetting;

 };
```

Role trigger

Role triggers invoke operations when a role is related or unrelated. You specify role triggers for a specific role in a specific relationship; it must be defined for the same relationship as the role to which it applies.

Trigger types

The operation must have one parameter, which is the type of the to entity defined in the role. The operation must be defined in the from entity.

Trigger type	Invokes the trigger operation when ...
relate	you relate instances using the role. The relate precedes the invocation of the trigger operation.
unrelate	you unrelate instances related through the role, explicitly or when the to instance is deleted. The unrelate follows the completion of the operation.

relate, unrelate The role trigger operation can be inherited. The role trigger operation must satisfy the following conditions:

- it is one-way, two-way, or a signal

- it is defined in the "from" entity
- its return type is void
- it has a single parameter of the "to" entity type, or the "associative" entity for associative relationships.

When two objects are related and one of them is deleted, the objects are implicitly unrelated. This invokes any unrelate trigger defined for the role “from” the remaining object. But if an unrelate trigger is defined “from” the deleted object, that trigger is not invoked by the implicit unrelate.

IDLos

Consider the following model in the Traffic package.

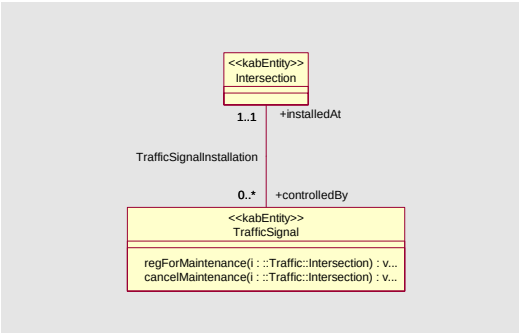

Figure 2.7. Relationship with operations for triggers

Use the trigger ... upon statement (for syntax details, see “trigger” on page 113.

Here is the IDLos for the example in Figure 2.7:

```
entity TrafficSignal
{
    oneway void regForMaintenance(in Intersection i);
    oneway void cancelMaintenance(in Intersection i);
};
entity Intersection
{
};
relationship TrafficSignalInstallation
{
    // roles
    role Intersection controlledBy 0..* TrafficSignal;
    role TrafficSignal installedAt 1..1 Intersection;

    // triggers
    trigger TrafficSignal::regForMaintenance
    upon relate controlledBy;
    trigger TrafficSignal::cancelMaintenance
    upon unrelate controlledBy;
};
```

Associative role triggers For associative relationships, you define the trigger operation in the “from” entity. This operation must take a single parameter whose type matches the associative entity.

Here is a modified version of the example for the marriage relationship presented in Figure 2.6.

Here is the modified example in IDLos:

```
entity Wife
{
    // new operation as target for the trigger
    void announceMarriage(in MarriageCertificate mc);
};

relationship Marriage
{
    Husband marries 1..1 Wife;
    Wife weds 1..1 Husband;
    using MarriageCertificate;
    trigger Wife::announceMarriage upon relate weds;
};

action Pkg::Wife::announceMarriage
{`
    // send out announcements
`};
```

Module

Modules define a namespace. Inside a package, modules represent the main means of defining namespaces.

You can add a module to one of the following:

- the section called “Package” on page 30
- Module

You can model anything in a module that you can in a package For a list of the elements see the section called “Package” on page 30.

IDLos

Use the module statement (for syntax details, see“module” on page 100).

Here is IDLos for the example above:

```
module myModule
{
   ...
};
```

Package

This section describes KIS packages and how to use them. Everything in KIS is organized into packages. Packages hide access to their entities, and each package is a namespace.

When you are implementing a package, you have open access to everything in your package. You also have open access to everything in other packages - except the entities. Entities are hidden inside

their package. The only way to use an entity in another package is through an interface (see the section called “Interface” on page 33).

Package containment

KIS packages may not be nested.

You can model all of the following within a package:

Element	Description
the section called “Entity” on page 11	Class of objects.
the section called “Local entity” on page 14	Non-distributable class of objects.
the section called “State machine” on page 38	Alias for a basic or composite type.
Const	Named constant (see “const” on page 301).
KIS types	Basic or composite type.
the section called “Relationship” on page 23	Defines an association between the same or two different entities. You can link instances of related entities through a relationship.
the section called “Inheritance” on page 42	Defines the inheritance relationship between two entities, the supertype and subtype; a subtype inherits the attributes and operations defined in the supertype.
the section called “Interface” on page 33 (and exposure)	Defines entity exposure across package borders.

Using Packages As a client of a package, you can:

- declare and create interfaces
- invoke exposed operations
- access exposed attributes
- navigate, relate, or unrelate exposed roles.
- implement abstract interfaces
- employ user-defined types

IDLos

Use the package statement (for syntax details, see“package” on page 105).

Here is the IDLos for the example above:

```
package PublicWorks
```

```
{
   ...
};
```

Example

Consider the PublicWorks package, which contains the following model, given first in its visual form, then in IDLos.

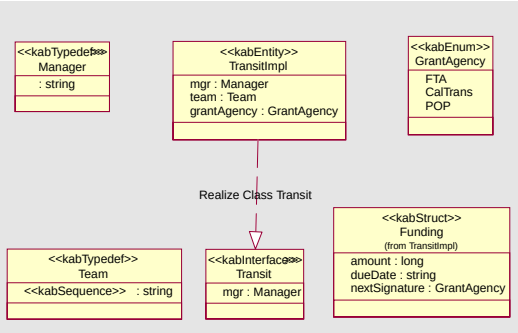

Figure 2.8. Package PublicWorks

The same model in IDLos:

```
package PublicWorks
{
    typedef string Manager;
    typedef sequence<string> Team;
    enum GrantAgency { FTA, CalTrans, POP };
    entity TransitImpl
    {
        struct Funding
        {
            long amount;
            string dueDate;
            GrantAgency nextSignature;
        };

        attribute Manager mgr;
        attribute Team team;
        attribute GrantAgency grantAgency;
    };
    interface Transit
    {
        attribute Manager mgr;
    };
     expose entity TransitImpl with interface Transit;
};
```

A client of the PublicWorks package would have access to the Manager, Team, Transit, and GrantAgency types. But the client would not have access to TransitImpl type nor to the Funding structure nested in the entity.

The Transit interface does give the client access to the mgr attribute of TransitImpl through Transit.mgr. Interfaces are discussed in detail below.

Interface

To give a client of your package access to an entity, you must provide an interface. The interface exposes the entity. The contents of the interface controls which attributes, operations, keys, and signals are exposed.

Each interface may expose only one entity. An entity may be exposed by more than one interface. Each operation or attribute in the interface must have a corresponding attribute, operation, or signal in the entity that it exposes.

Exposing types

An interface is an alias of the entity it exposes. You can use an interface type wherever you may use the entity type.

The following table shows the elements of an entity that an interface can expose:

Element	Description
the section called " Attribute" on page 15	Attribute in the entity that the interface exposes.
the section called " Key" on page 22	Key in the entity that the interface exposes.
the section called "Operation" on page 16	Operation in the entity that the interface exposes.
the section called " Signal" on page 41	Signal in the state machine of the entity that the interface exposes.

Namespace The namespace constraints that apply to entities also apply to interfaces (see the section called "Namespace" on page 12).

Association The association constraints that apply to entities also apply to interfaces (see the section called "Relationships" on page 12).

Instantiation Instantiating an interface creates an object of the exposed entity type. You instantiate an interface with the same action language statements used to instantiate entities. (See "Creating objects" on page 126 for details.)

Exposure The following table lists the elements that you can expose with an interface and shows how each element is exposed:

To expose ...	You must ...
an attribute	declare the same attribute in the interface (see the section called " Attribute" on page 15). The attribute must match the entity's attribute in name and type. Attribute access can also be marked `readonly` in an interface, exposing only the get accessor outside the package.

To expose ...	You must ...
a key	declare the key just like it is declared in the underlying entity.
a signal	declare a oneway void operation of the same name and signature as the signal in the entity (see the section called " Signal" on page 41).
an operation	declare an operation having the same name, signature (see the section called "Operation" on page 16), and return type as the one in the entity. It must also match in its `local` properties. Operations marked `oneway` in an entity must also be marked `oneway` in an exposing interface.
a relationship between entities	define it between interfaces (see the section called " Relationship" on page 23). Do not define the same relationship between the corresponding entities; defining it between the interfaces also defines it for the entities.
user-defined types (such as structs, enumerations, and typedefs) defined in an entity	move the type definition from the entity to one of its interfaces (on types, see Chapter 2). A type defined in an entity is hidden from other packages. A type defined in package scope, module scope, or an interface scope is automatically exposed to other packages. Types defined in an interface are exposed, just as everything else defined in the interface is exposed.

Interfaces and adapters KIS adapter factories can generate different code for the same interfaces. This is why more than one interface per entity is allowed.

In the following model, three interface expose the same entity, one for each of the technologies SNMP, CORBA and other.

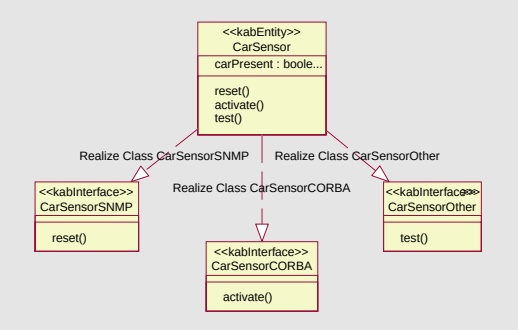

Figure 2.9. Interface in an UML diagram

Here is the IDLos for the same model:

```
entity CarSensor
{
    void reset();
    void activate();
    void test();
};
interface CarSensorForSNMP
{
    void reset();
};
interface CarSensorCORBA
{
    void activate();
};
interface CarSensorOther
{
    void test();
};
expose entity CarSensor with interface CarSensorSNMP;
expose entity CarSensor with interface CarSensorCORBA;
expose entity CarSensor with interface CarSensorOther;
```

In the Design Center, the adapter could generate SNMP agent code for reset, CORBA server code for activate, and code for another technology for the test operation.

IDLos

Use the interface statement (for syntax details, see “interface” on page 93).

Here is the IDLos for the example above:

```
interface TimerEvent
{
    attribute ::Traffic::Road road;
    void generate();
};
```

To expose an entity with an interface, use the expose statement (for syntax details, see “expose” on page 92).

For example, to expose the entity TimerEventImpl with the TimerEvent interface, defined above, use the following IDLos:

```
expose entity TimerEventImpl with interface TimerEvent;
```

Interface properties

An interface can be tagged by properties that affect exposure. Also, an interface can expose a singleton entity only if it is a singleton and a local entity only if it is a local interface.

The following table shows the properties you can use in the definition of an interface.

Property	Description
abstract	Prohibits the interface from exposing an entity (see the section called “ abstract” on page 36).

Property	Description
createaccess	Permits and revokes permission to instantiate the exposed entity through the interface (see the section called “Access control” on page 37).
deleteaccess	Permits and revokes permission to delete instances of the exposed entity through the interface (see the section called “Access control” on page 37).
extentaccess	Permits and revokes permission to access the extent of the exposed entity through the interface (see the section called “Access control” on page 37).
singleton	If an interface exposes a singleton, it must also have the `singleton` property (see the section called “ singleton” on page 13).
local	If an interface exposes a local entity, it must also have the `local` property (see the section called “ Local entity” on page 14 and the section called “Local Interface” on page 37)

Setting interface properties In IDLos interface properties appear before the interface statement in square bracket, for example,

```
[ abstract ]
interface TalkBack
{
    oneway void respond(in string response);
};
```

abstract Abstract interfaces define the operations and attributes which a client package should both inherit and implement. In other words, abstract interfaces define callbacks or notifiers.

Abstract interfaces can be used in an inheritance hierarchy. For example the following is legal:

```
[ abstract ]
interface Top
{
    attribute long    name;
};

[ abstract ]
interface Bottom : Top { };
```

The base interface in the inheritance hierarchy must be [abstract].

An entity that implements (is exposed by) an abstract interface may not implement any other interfaces.

Abstract interfaces that are not explicity implemented may be implemented by inheriting from them with another interface and then implementing the child interface with an entity. For example:

```
[ abstract ]
 interface Abstract
```

```
{
    attribute long  aLong;
};

interface Concrete : Abstract { };
entity ConcreteImpl
{
    attribute long  aLong;
};

expose entity ConcreteImpl with interface Concrete;
```

The entity implementing the interface does not have to be a base entity. It can be an entity participating in an inheritance hierarchy. For example the following is legal IDLos:

```
[ abstract ]
interface I
{
     void anOp();
};

interface IX : P::I { };

entity Base { };
entity Child : Base
{
     void anOp();
};

expose entity Child with interface IX;
```

the section called “A big example” on page 53 contains an example of defining and using abstract interfaces.

Access control To permit more control over how an entity is exposed, three properties grant or revoke access to create, delete, or retrieve the extent of an interface. In this example, a client can retrieve handles to all the Crosswalks instantiated, but the client cannot create or delete any of them.

```
[
    createaccess=revoked,
    deleteaccess=revoked,
    extentaccess=granted
]
interface Crosswalk {};
```

In this example, new BankCustomers can be created. But for security, clients cannot search through the existing customers, and they cannot delete any customers.

```
[
    createaccess=granted,
    deleteaccess=revoked,
    extentaccess=revoked
]
interface BankCustomer {};
```

Local Interface

To give a client of your package access to a local entity, you must provide an interface. The local interface exposes the local entity. The contents of the interface controls which operations are exposed.

To expose an operation, the local interface defines the same operation.

The following table shows the elements of a local entity that a local interface can expose:

Element	Description
the section called “Operation” on page 16	Operation in the entity that the local interface exposes.

IDLos

Use the interface statement (A local interface is an interface tagged with the local property (for syntax details, see “interface” on page 93).

```
[local]
entity UtilitiesImpl
{
    void who();
    void whatis();
};
[local]
interface Utilities
{
    void who();
    void whatis();
};
```

To expose a local entity with a local interface, use the expose statement (for syntax details, see “expose” on page 92).

For example, to expose the entity UtilitiesImpl with the Utilities interface, defined above, use the following IDLos:

```
expose entity UtilitiesImpl with interface Utilities;
```

State machine

State machines define the stages of an instance’s life as well as the transitions between those stages. Use them when the instances of an entity must perform duties in a certain order, or when an instance must wait for certain signals before it can proceed. State machines are the best way to manage asynchronous communication with another instance or with a network protocol.

You can define a finite state machine for an entity. An action is executed as an instance enters each state. There are no actions associated with the signals or transitions. The actions are implemented in action statements using action language (on action language, see Chapter 3).

The following table shows the modeling elements that you use to model an entity’s state machine:

Element	Description
the section called “ State” on page 40	A condition or situation during the life of an instance. An instance remains in a state until it receives a signal.
the section called “ Signal” on page 41	An event causing a state transition.

Element	Description
the section called “ Transition” on page 41	A progression from one state to another, or the same, state caused by a signal.

IDLos

An UML representation of a state machine is shown in the following figure.

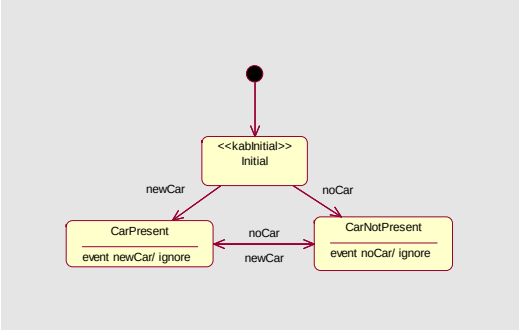

Figure 2.10. State model for car entity in sample model

A state machine consists of a stateset statement and a group of transition statements, within the scope of the entity.

The stateset statement specifies all the states valid for the entity as well as the initial state and final states (for syntax details, see “stateset” on page 110).

A transition statement specifies the from state and the to state as well as the signal initiating the transition.

```
entity CarSensor
{
    // attributes
    attribute boolean carPresent;

    // states
    stateset
    {
        Initial,
        CarPresent,
        CarNotPresent
    } = Initial;

    // signals
    signal newCar( );
    signal noCar( );

    // transitions
    transition Initial to CarPresent upon newCar;
    transition Initial to CarNotPresent upon noCar;
    transition CarPresent to CarNotPresent upon noCar;
    transition CarNotPresent to CarPresent upon newCar;
    transition CarPresent to ignore upon newCar;
    transition CarNotPresent to ignore upon noCar;
};
```

State

You use states to model state machines (see the section called " State machine" on page 38). A state represents a stage in the life of an instance. An instance stays in the same state until it receives a signal causing it to tranistion to a different state.

There are two special types of states:

- initial state
- final state

There can only be one initial state. This is the state that an instance first enters upon creation.

The state action for the initial state is executed following a transition back to the initial state, but not when the object is created. Actions on create can be expressed using create triggers—see the section called "Entity trigger" on page 25.

There is no limit to the number of final states. When an instance enters a final state, it ceases to exist; the runtime deletes the instance.

Implementation of states

This section describes the implementation of state actions.

Actions You implement operations and states in action statements.

```
action ::Traffic::CarSensor::CarNotPresent
// arguments: None
{`
    self.carPresent = false;
`};
```

Actions specify the name of an operation or a state. They can be defined inside an entity, outside the entity, or even outside the package, so use a scoped name when needed. The action language that implements the action appears between the backquotes (‘) in IDLos. See Chapter 7 for a description of the KIS action language.

You must define an action statement for each operation and state. Although empty actions are legitimate, forcing you to write them explicitly reminds you to define an action where one is needed.

Everything between the backquotes is handled by a different parser from the rest of IDLos. This is why the IDLos namespace does not have reserved words from the action language, and vice-versa.

No parameter signature is required with the action statement.

Actions can be defined outside of packages so you can organize them into separate files, if you choose.

IDLos

The stateset statement defines all the states for an entity, as well as the initial and final states (for syntax details, see "stateset" on page 110).

The initial state is the state that appears after the equals sign in the stateset statement.

finished This property designates the terminal state(s) in a stateset. It requires a list of states, for example:

```
[finished = {Retired, Lost}]
stateset {Made, Used, Retired, Lost} = Made;
```

In this example, KIS automatically deletes the object after the Retired or Lost state finishes executing its action.

Signal

A signal causes a transition to a new state.

Only in parameters are allowed. A signal does not have a raises clause.

The signal statement defines the name of the signal and an optional list of parameters.

```
signal newCar( );
signal noCar( );
signal carLost(in string registration);
```

Transition

A transition specifies that an instance moves from one state to another when a certain signal is received. in the source state will perform certain specified actions and enter the destination state when a specified event occurs or when certain conditions are satisfied. A state transition is a relationship between two states, two activities, or between an activity and a state.

```
transition Initial to CarPresent upon newCar;
transition Initial to CarNotPresent upon noCar;
transition CarPresent to CarNotPresent upon noCar;
transition CarNotPresent to CarPresent upon newCar;
```

The first transition above says that if an object is in the Initial state and the newCar signal is received, transition to the CarPresent state and execute its action.

If a signal is received, and there is no transition from the current state for that signal, a KIS system exception is thrown by default. You can explicitly express the default behavior by specifying a transition to cannothappen.

```
transition CarPresent to cannothappen upon lostCar;
```

The transition cannothappen (whether expressed explicitly or by default) indicates a condition that should never be possible in your model. Do not use this transition for expected error conditions.

To ignore signals when you are in a certain state, specify a transition to ignore.

```
transition CarPresent to ignore upon newCar;
transition CarNotPresent to ignore upon noCar;
```

If two or more transitions to the same state occur upon different signals, those signals must all have the same parameter signature.

Inheritance

Like most object-oriented languages, KIS supports inheritance. This section describes how inheritance works in KIS and discusses

- the section called “Entity inheritance” on page 42
- the section called “ Operations” on page 44
- the section called “ Interface inheritance” on page 49

Entity inheritance

KIS entities may inherit from other entities. This means that the subtype can share all the features of the supertype entity. The subtype inherits from the supertype.

The following constraints apply:

- a subtype may have only one supertype
- an entity may inherit from another entity if they are in the same package
- local entities may inherit only from other local entities

What do subtypes inherit? A subtype inherits these items:

- attributes
- operations; operations will be discussed in detail below
- roles; any role you can navigate, relate or unrelate on a supertype, you can navigate, relate or unrelate through a subtype
- state machines; subtypes inherit all signals, transitions; all states and their actions; a subtype cannot override the state machine of its supertype
- keys
- lifecycle (create/delete) and attribute triggers

What can subtypes add? A subtype may add:

- attributes, with triggers if desired
- keys
- operations, both virtual and non-virtual
- a state machine, if it did not inherit one; you cannot add to an existing state machine in a subtype; the state machine must be defined entirely within one entity.

IDLos In UML diagrams, inheritance is represented with a solid line from the subtype to the supertype. The tip of the line is a large hollow triangle pointing to the supertype.

For example, in the following model the subtype RoadSide1 inherits from the supertype SideOfRoad (see the figure below). This means that all instances of RoadSide1 will share the attribute direction and its accessors. It also means that anywhere you can use the type SideOfRoad, you can use the type RoadSide1.

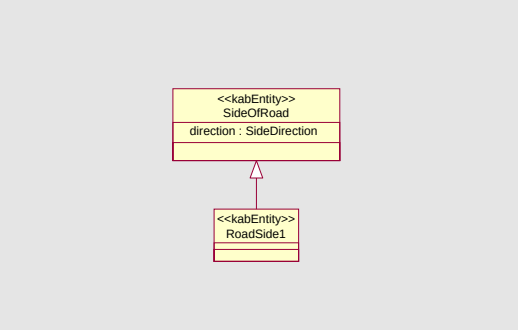

Figure 2.11. RoadSide1 inherits from SideOfRoad

In IDLos, use the colon to establish an inheritance relationship between two entities: subtype : supertype.

Here is the IDLos for the model defined in Figure 2.11:

```
entity SideOfRoad
{
    attribute SideDirection direction;
};
entity RoadSide1 : SideOfRoad
{
};
```

Shadow types Nested types defined in a supertype are accessible to a subtype. Consider the model where struct B is defined within the scope of entity A; entity C inherits from entity A; struct B is accessible inside entity C, for example, as a type in an attribute definition of b.

This situation is represented in the following model

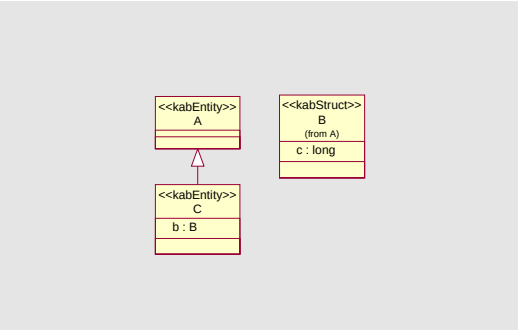

Figure 2.12. Inheritance and shadow types

Here is the same model in IDLos:

```
entity A
{
    struct B
    {
        long c;
    };
};
entity C : A
{
    attribute B b;
};
```

The example shows how entity C can use type B without any scoped name. These inherited, nested types are called shadow types.

Operations

All regular (non-virtual) operations are inherited: you can invoke the supertype's operation through the subtype. The supertype's implementation of the operation will be executed.

In the next model the entity Supertype has an operation named greeting. The implementation of greeting is to print "Hello" to stdout.

```
printf("Hello\n");
```

The entity Subtype inherits from Supertype, as shown in the figure below.

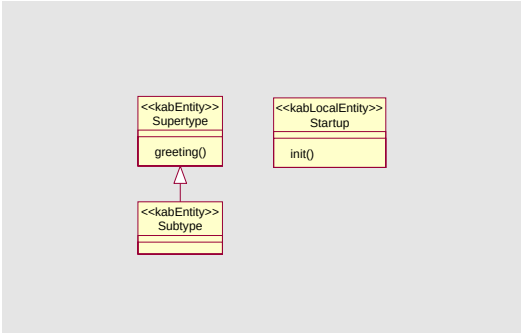

Figure 2.13. Supertype and Subtype

There is a third entity in the model called Startup that has an operation called init. init is a lifecycle operation that the runtime invokes when the component is initialized. The implementation of init is to create an instance of Subtype and invoke the operation greeting on the instance.

```
declare Subtype sub;
create sub;
sub.greeting();
```

Invoking greeting on the instance of Subtype sends "Hello" to stdout.

Here is the same model defined in IDLos:

```
// define the supertype
entity Supertype
{
```

```
    void greeting();
    action greeting
    {‘
        printf("Hello\n");
    ‘};
};

// define the subtype
entity Subtype: Supertype
{};

// A local entity to start the engine
[local] entity Startup
{
    [initialize]
    void init();
};
action Startup::init
{‘
    // invoke through the subtype
    declare Subtype sub;
    create sub;
    sub.greeting();
‘};
```

Redefining operations When an inherited operation is redefined in a subtype, KIS normally invokes either the supertype's or subtype's implementation, depending on the type of the object handle.

In the following model, the operation parting is defined in the supertype and its subtype, as shown in the figure below.

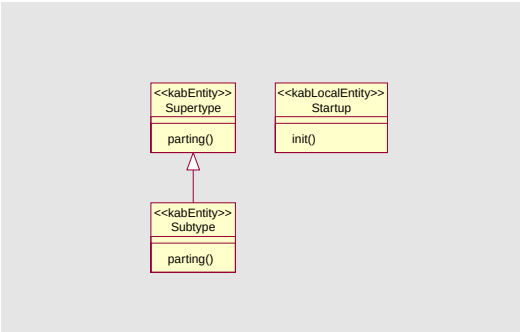

Figure 2.14. Supertype and Subtype

In the Supertype, parting sends “Bye ...” to stdout and in Subtype parting sends “Cheers ...”.

init declares an object handle of type Supertype and one of type Subtype. It creates an instance of Subtype and assigns it to the Supertype. Finally, it invokes parting first on the supertype instance an then on the subtype instance.

Here is the action language implementation of init:

```
    // Create subtype
    declare Subtype sub;
    create sub;
    // Make a supertype handle to the subtyped object
    declare Supertype super;
    super = sub;     // these now refer to the same object
```

```
    // Invoke via supertype and subtype
    super.parting();
    sub.parting();
```

The result of invoking init is “Bye...Cheers...”. Calling super.parting() prints “Bye...”, and then calling sub.parting() prints “Cheers...”.

Here is the same model in IDLos:

```
// define the supertype
entity Supertype
{
    void parting();
    action parting
    {‘ printf("Bye..."); ‘};
};

// define the subtype
entity Subtype: Supertype
{
    void parting();
    action parting
    {‘ printf("Cheers...");‘};
};
[local] entity Startup
{
    [initialize]
    void init();
};

action Startup::init
{‘
    // Create subtype
    declare Subtype sub;
    create sub;

    // Make a supertype handle to the subtyped object
    declare Supertype super;
    super = sub;     // these now refer to the same object

    // Invoke via supertype and subtype
    super.parting();
    sub.parting();
‘};
```

When executed, this model prints “Bye...Cheers...”. Calling super.parting() prints “Bye...”, and then calling sub.parting() prints “Cheers...”.

Virtual operations and polymorphic dispatch Sometimes you don’t want to execute the supertype’s implementation—you want to use an object as a supertype, but when you call an operation, you want to invoke the subtype implementation that corresponds to the actual object. This is called polymorphism.

IDLos’s virtual property provides polymorphic dispatching of KIS operations to the correct subtype implementation. The following example shows how virtual changes the way operations are dispatched. The preceding example printed “Bye...Cheers...” but this one prints “Cheers...Cheers” because it uses virtual.

In the following example, the parting operation is declared virtual in the Supertype entity. When init executes it invokes the subtype’s implementation twice because both object handles reference a subtype object.

In IDLos, you make the same change to the model by prepending the parting operation definition in the Supertype entity with [virtual], as shown below:

```
// define the supertype
entity Supertype
{
    [virtual] void parting();
    action parting
    {‘ printf("Bye\n"); ‘};
};
```

Local entities cannot contain virtual operations.

You can use the virtual property at any level of an inheritance hierarchy, but you can’t use it again further down the hierarchy.

In the following example (see the figure below), anOp is virtual in Middle and a polymorphic dispatch will occur on instance handles of Middle and Bottom, but not Top.

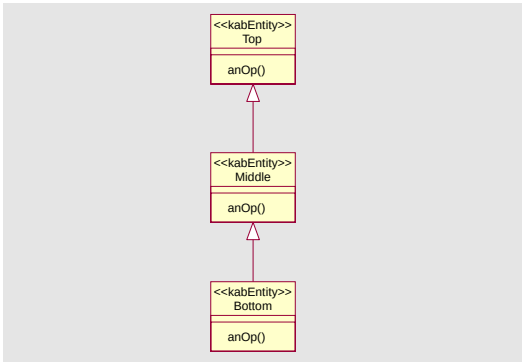

Figure 2.15. Polymorphic dispatch example

Here is the same model in IDLos, which shows the virtual property:

```
entity Top
{
    void anOp();
};
entity Middle : Top
{
    [virtual]
    void anOp();
};
entity Bottom: Middle
{
    void anOp();
};
```

Adding an entity Floor that inherits from Bottom and then declaring anOp as virtual in Floor will create an error, since virtual has already been specified on the Middle level.

Don’t try to make a subtype call the supertype implementation of a virtual operation. The supertype always invokes the subtype’s implementation. This has important consequences: for the example above, the following implementation will cause an endless loop at runtime.

```
action Bottom::anOp
{‘
    declare Middle mid;
    mid = self;// upcast
    mid.anOp(); // will recurse forever
‘};
```

Instead of trying to invoke the virtual operation in the base entity directly, you can make it easy for subtypes to use the supertype’s implementation using the following style:

```
entity Base
{
    string getDefaultString();
    [virtual] string getString();
};
entity Child : Base
{
    string getString();
    action getString
    {‘
        declare Base base;
        base = self;
        return base.getDefaultString();
    ‘};
};
```

In this way, the base class provides both a default implementation and an operation that may be overridden by subtypes.

pure virtual operations You can use the `pure` property to force derived types to provide implementations for a virtual operation. A pure virtual operation has no implementation in the base type. For example:

```
package y
{
    interface IBase
    {
        [virtual, pure]
        void anOp();
    };
    entity IBaseImpl
    {
        [virtual, pure]
        void anOp();
    };
    expose entity IBaseImpl with interface IBase;
};

package x
{
    interface IChild : A::IBase
    {
    };
    entity IChildImpl
    {
        void anOp();
        action anOp
        {`
            printf("IChildImpl::anOp called\n");
        `};
    };
};
```

```
package d
{
    [ local ]
    entity Startup
    {
        [ initialize ]
        void init();
        action init
        {`
            declare y::IBase        ib;
            declare x::IChild       ic;

            // virtual dispatch
            create ic;
            ib = ic;
            ib.anOp();

            // causes runtime exception, no impl exists
            create ib;
            ib.anOp();
        `};
    };
};
```

The `pure` property can only be used together with the `virtual` property.

Interface inheritance

Interfaces can inherit from other interfaces in the same package or in other packages. The subtype inherits all interface definitions from the supertype.

Generally, the following rules apply to new definitions. The interface subtype can:

- introduce new types
- expose additional attributes
- expose additional operations
- restrict control access, but cannot grant more

Operations and attributes must expose an entity that is a subtype of the entity exposed by the supertype (where an entity is considered a subtype of itself)

Inheritance within a package When the supertype and subtype interfaces are in the same package, the subtype interface must expose an entity that is a subtype of the entity exposed by the supertype interface, where an entity is considered a subtype of itself.

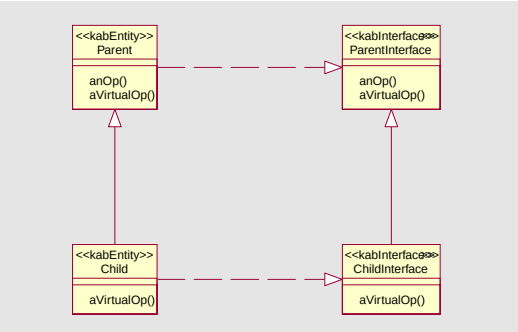

Figure 2.16. Interface inheritance

The model illustrated in Figure 2.16 is expressed in IDLos as follows:

```
package myPackage
{
    entity Parent
    {
        void anOp ();
        [ virtual ]
        void aVirtualOp ();
    };
    entity Child: Parent
    {
        void aVirtualOp ();
    };
    interface ParentInterface
    {
        void anOp ();
        [ virtual ]
        void aVirtualOp ();
    };

    interface ChildInterface: ParentInterface
    {
        void aVirtualOp ();
    };
    expose entity Parent with interface ParentInterface;
    expose entity Child with interface ChildInterface;
};
```

Cross-package inheritance When the supertype and subtype interfaces are in different packages, KIS cannot validate operation redeclarations through the exposed entities; in KIS cross-package inheritance relationships do not exist between entities. For this reason, it is illegal to redefine or redeclare an operation in the subtype.

Namespaces

KIS namespaces are hierarchical, with packages forming the outermost namespace. Within a package, you can use modules to impose additional namespaces when needed. Within packages and modules, many KIS elements define a namespace.

All identifiers in a namespace must be unique. For example, an entity defines a namespace, so it cannot contain an operation and an attribute with the same name.

Modules

Modules provide extra namespaces within a package if needed.

Modules can be nested. A module may contain entities, interfaces, enums, structs, exceptions, typedefs, nested modules, relationships, actions, and exposes statements.

When a module is declared more than once in IDLos, the module is re-opened and modified on each subsequent declaration. More information will be added to its definition each time that the module name appears in the model.

Model elements defining namespaces

Package and module are not the only containers that form a namespace. The following table includes all the IDLos language elements that form a namespace.

This namespace...	...may contain these namespaces
package	module, entity, interface, struct, exception, relationship
module	module, entity, interface, struct, exception, relationship
interface	struct, operation, exception
entity	struct, operation, exception, signal
struct	struct
operation	
signal	
relationship	role

Scoped names

In IDLos and action language, to identify an element in its namespace, use the scoped name. The scoped name is formed by listing each namespace of the hierarchy, separated by a pair of colons (::).

For example, consider the model in the following example:

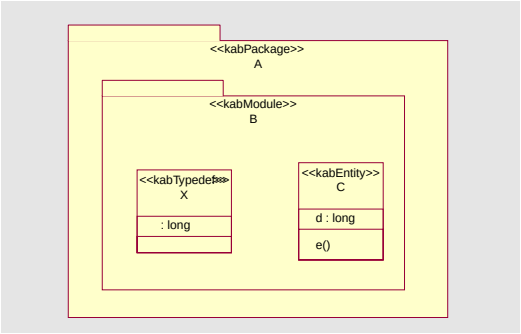

Figure 2.17. Namespace example

Here is the same model in IDLos:

```
package A
{
    module B
    {
        typedef long X;
        entity C
        {
            attribute long d;
            void e();
        };
    };
};
```

To refer to the attribute in the example, you would say

```
::A::B::C::d
```

The leading (::) mean the name is globally scoped. Globally scoped names are like absolute addresses. They tell IDLos to start resolving the name outside of any package scope (namespace).

A relative address is partially scoped. Partially scoped names are resolved starting from where they are used. For example, when you implement operation ::A::B::C::e, you are in e's namespace. Here are three ways to refer to the typedef X from within such an action:

```
action ::A::B::C::e
{‘
    declare ::A::B::X aa;    // globally scoped
    declare A::B::X bb;    // partially scoped
    declare B::X cc;    // partially scoped
    declare X dd;    // unscoped
‘};
```

A partially scoped name is searched for outwards from each enclosing scope. So in the example, the second declaration (A::B::X) starts a search in namespace e. Within e, it does not see a namespace or name A. So it goes outward to the next namespace, C. Within C, it does not see a namespace or name A, so it goes outward to B. Within B, it does not see an A. Out again to the global scope. In the global scope, it sees A. Within A, it sees B. Within B it sees X. The name is resolved.

Ordering and forward declarations in IDLos

IDLos is a one-pass parser. This means the parser must see the definition of something before it is used. This is true even within an entity. For instance, you must define the signals before they are used in transition statements.

Entities and interfaces can be forward declared (this is explained in the section called “A big example” on page 53). Forward declarations satisfy the parser: having seen the name, the parser lets you use the entity or interface.

```
// forward declares
entity A;
entity B;
interface C;

relationship R
{
    role A owns 0..* B;
```

```
};
entity A
{
    attribute C c;
    attribute D d;     // error!  D not defined yet!
};
typedef long D;
```

The action language is not parsed until after the entire model is loaded into the Design Center and you request a build. So you don't have to worry if the types you refer to in your action are defined below or above the action in the IDLos file. Once they have all been loaded, the types will already be in the Design Center model sources and will be available to the entire model.

A big example

Here is a larger example model. It is an example of the notifier pattern. Use this pattern when you define "callbacks" or "notifiers" in your server component. It demonstrates the use of interfaces, inheritance and exposure:

- An abstract interface is defined in the server package.
- An interface in the client package inherits from the abstract interface.
- The subtype interface in the client package exposes an entity in the client.

At runtime the server can access a callback in the client through the abstract interface.

The example defines a server package and a client package. The server listens for a message from the client; if the server receives "Hello" then it replies "Bonjour", and replies "I don't understand" to any other message. When the client receives the server's response, it prints "Ok" or "Not Ok".

UML

The following graphic shows the HelloWorldSever package as an UML diagram.

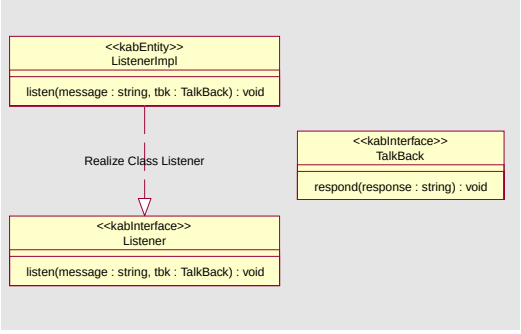

Figure 2.18. HelloWorldServer

The interface Listener exposes the operation listen in the entity ListenImpl.

The following action language implements the operation listen:

```
if (message == "Hello")
{
    tbk.respond(response:"Bonjour");
```

```
}
else
{
    tbk.respond(response:"I don't understand");
}
```

Also, the abstract interface TalkBack exposes the operation respond.

The following graphic shows the HelloWorldClient UML diagram.

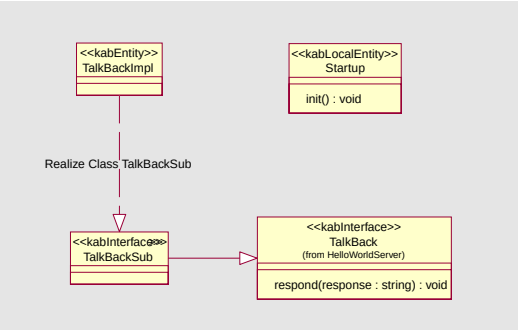

Figure 2.19. HelloWorldClient

The interface TalkBackSub inherits from the abstract interface TalkBack defined in the HelloWorld-Server package and exposes the entity TalkBackImpl.

The operation init in the local entity Startup is the life cycle operation that kicks off the application. It creates an instance of TalkBackSub and Listener from the server package and invokes listen on the Listener instance passing the instance of TalkBackSub as the callback object.

The follow action language is the implementation of init:

```
declare ::HelloWorldServer::Listner lis;
declare TalkBackSub tks;
create lis;
create tks;
lis.listen(message:"Hello", tbk:tks);
```

Through the inherited abstract interface TalkBack, TalkBackSub exposes the signal respond defined in TalkBackImpl. The graphic below shows the state machine for the entity TalkBackImpl.

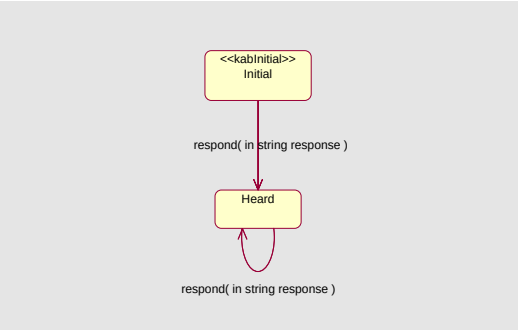

Figure 2.20. TalkBackImpl state machine

When an instance of TalkBackImpl receives the signal respond, it invokes the action language defined for the state Heard.

The following action language implements the state Heard:

```
if (response == "Bonjour")
{
    printf("Ok\n");
}
else
{
    printf("Not Ok\n");
}
```

IDLos

Here is the IDLos for the same model:

```
[annotation= "This server says 'Bonjour' when clients say 'Hello'"]
package HelloWorldServer
{
    // forward declares
    interface TalkBack;

    [annotation= "This entity does all the work."]
    entity ListenerImpl
    {
        [
            annotation= "The TalkBack reference must be passed in"
            " by the client. The server uses it to respond."
        ]
        oneway void listen(in string message, in TalkBack tbk);
    };

    [
        annotation= "This interface exposes the ListenerImpl”
                    “entity. It allows create and delete access”
                    “so that clients don’t need a factory.",
        createaccess=granted,
        deleteaccess=granted
    ]
    interface Listener
    {
        oneway void listen(in string message, in TalkBack tbk);
    };
    expose entity ListenerImpl with interface Listener;

    [   annotation=
            "This is the notifier. Because clients inherit from ”
            “this interface (and implement it), it must be “
            “abstract. We also grant access, because clients’ “
            “subtype of this notifier will need it",
        createaccess=granted,
        deleteaccess=granted,
        abstract
    ]
    interface TalkBack
    {
        oneway void respond(in string response);
    };

    action ListenerImpl::listen
    {‘
```

```
        if (message == “Hello”)
        {
            tbk.respond(response:”Bonjour”);
        }
        else
        {
            tbk.respond(response:”I don’t understand”);
        }
    ‘};
};

//
// Now the client.
//
package HelloWorldClient
{
    [
        annotation=
            "Let’s implement the notifier. This is the key to”
            “the notifier pattern: inheriting the abstract”
            “interface, and implementing the derived interface"
            "in an entity. No operation declaration is needed in”
            “the TalkBackSub - it is inherited."
    ]
    interface TalkBackSub : ::HelloWorldServer::TalkBack
    {};

    [annotation=”Process the response in a small state machine."]
    entity TalkBackImpl
    {
        signal respond(in string response);
        stateset {Initial, Heard} = Initial;
        transition Initial to Heard upon respond;
        transition Heard to Heard upon respond;

        [annotation="Implements the state Heard."]
        action Heard
        {‘
            if (response == "Bonjour")
            {
                printf("Ok\n");
            }
            else
            {
                printf("Not Ok\n");
            }
        ‘};
    };
    expose entity TalkBackImpl with interface TalkBackSub;

    [
        annotation=
            "The operation in this native entity with the engine"
            "event property starts the client in motion.",
        local
    ]
    entity StaRtup
    {
        [initialize]
        void init();
        action init
        {‘
            declare ::HelloWorldServer::Listner lis;

            // It is important to instantiate a TalkBackSub,
            // rather than a TalkBack. In the type
            // hierarchy, TalkBackSub is both a TalkBack
```

```
            // and a TalkBackImpl.
            declare TalkBackSub tks;
            create lis;
            create tks;
            lis.listen(message:"Hello", tbk:tks);
        ‘};
    };
};
```

3

Action language

This chapter describes the Kabira Infrastructure Server (KIS) action language. The action language is used within `action` statements in IDLos. IDLos is the KIS modeling language discussed in Chapter 2. Before reading this chapter, you should already be familiar with object-oriented software development, and you should have read the modeling language chapters of this book.

This chapter contains the following sections:

Overview

In KIS, most of your application’s behavior will be implemented by action language statements.

Action language and IDLos

Action language is the dynamic counterpart to the IDLos structural modeling language. You use IDLos to design the entities, relationships, and states of your model. You use action language to define what happens during these states (the state actions) and what happens when the operations are invoked. You can also call C++ functions from your action language, or even include C++ code in-line.

Why is there an action language?

Suppose you have defined your entity model and your state model using IDLos. This model describes a large part of your application. But you still need to describe what happens in each state, and what your operations do.

Action language lets you specify actions that implement the operations and state behavior of your model. But unlike regular programming languages, action language lets you specify this behavior at a very high level of abstraction. Instead of dealing with memory allocation and index tables, you do things like selecting objects or traversing relationships.

Action language

Modeling in KIS is at a very high level of abstraction, higher than traditional object-oriented languages. For instance, you can define relationships between objects. You can define object state machines to handle asynchronous protocols. You can write queries to find objects, or to filter extents.

However, object models are not enough to implement an application, so KIS has an action language. You implement your models at a high level of abstraction using action language.

What is action language like?

The KIS action language uses a similar syntax to that of C++, and Java. If you know one of these languages, many features of the action language are already familiar to you. But action language is simpler, because of its higher level of abstraction. KIS action language looks like this:

```
declare short currentCall;
for( currentcall=0; currentCall<maxCalls; currentCall++ )
{
    if( thisCustomer.callsToday == 0 )
    {
        break;
    }
    thisCustomer.enterServiceCall( currentCall );
}
```

Some basic features of the action language

This section describes a variety of basic features of action language. Object manipulation is described in the following section.

Keywords

There is a section on each action language statement at the end of this chapter. Each of those statements is a keyword in action language. Additionally, all C++ keywords are also reserved in action language:

asm	for	static
auto	friend	static_cast
bool	goto	struct
break	if	switch

case	inline	template
catch	int	this
char	long	throw
class	mutable	true
const	namespace	try
const_cast	new	typedef
continue	operator	typeid
default	private	typename
delete	protected	union
do	public	unsigned
double	pure	using
dynamic_cast	register	virtual
else	reinterpret_cast	void
enum	return	volatile
explicit	short	wchar_t
extern	signed	while
false	sizeof	

Preprocessor directives Like IDLos, action language supports the `#include` preprocessor directive. But action language also supports the `#pragma include` directive, which includes a file after code generation and before C++ compilation. This allows you to include C++ header files for external libraries:

```
#pragma include <myCplusplusHeaders.h>
someCplusplusFunction();
```

Variables

Declaration Variable declarations define the name and type of a local variable. You must declare action language variables before using them:

```
declare type name;
```

You can also declare variables with initial values, or declare them as constants:

```
declare long x = 0;        // declaration with initial value
declare const long x = 0;  // declaration of a constant
```

Strings can be declared as both bounded and unbounded.

```
//
//  Declare an unbounded string
//
declare string    unboundedString;

//
//  Declare a bounded string of 100 bytes
//
declare string < 100 >    boundedString;
```

In state actions, action language variables must not have the same name as parameters to the signal(s) that transition to that state. Similarly, where an action implements an operation, action language variables must not have the same name as parameters to the operation. For example:

```
signal cursed (in long howManyYears);
transition alive to zombie upon cursed;
action zombie
{‘
    declare long howManyYears; // invalid! Is parameter name
‘};
```

You can also include C++ variable declarations in your action language. This lets you use variable types not provided by the action language. However, these types may be incompatible with IDLos types and the auditor cannot check them. For example:

```
action ::adventure::xyzzy
{`
    declare long longVar1;
    long    longVar2;
    struct Stat statBuf;

    longVar1 = 17;
    // legal, since the types are compatible
    longVar2 = longVar1;

    // Type mismatch here, but the auditor can’t see it.
    // Instead the C++ compiler will generate an error
    // and that is harder for the developer to deal with
    statBuf = longVar2;
 }`
```

Scope and lifetime When a variable is declared in an action; its scope is simply the scope of that action. When the action is completed, the variable ceases to exist:

```
action oneAction
{‘
    declare long myVariable;
    myVariable = 5;
‘};
action otherAction
{‘
    declare long anotherVariable;
    anotherVariable = myVariable;
    // that was illegal: “myVariable” is not in this scope
‘};
```

Code blocks form an inner scope. When a variable is declared within an action; its scope is the code block itself. When the code block is finished, the variable ceases to exist:

```
declare long myOuterVar;
while (x > 0)
{
    declare long myInnerVar;
    // do something
}
myOuterVar = myInnerVar;   // illegal - out of scope
```

Also, it is illegal to redefine a variable within an inner code block:

```
declare long myVar;
```

```
while (x > 0)
{
    declare long myVar;   // illegal - redeclared in scope
    ...
}
```

Accessing signal parameters from a state action In a state action, you can access parameters to the signal that caused the transition. For example:

```
signal cursed (in long howManyYears);
transition alive to zombie upon cursed;
action zombie
{‘
    declare long zTime;
    for( zTime=0; zTime < howManyYears; zTime++ )
    {
        // be a zombie for another year...
    }
‘};
```

Object references You can declare a variable using the name of an entity in the package. This creates an empty object reference (also called a handle) that you can create or assign objects to:

```
declare
{
    Customer thisCustomer;
    Customer thatCustomer;
}
create thisCustomer;       // create a new object
thatCustomer=thisCustomer;
// thisCustomer and thatCustomer refer to the same object
// This assigned an object reference, is not a "deep" copy
```

Manipulating data

Assignment Assignment statements give values to a variety of expressions in action language. They can contain:

- declared variables
- literals
- attributes on declared objects
- operations on declared objects
- return values from operations on declared objects

For example:

```
myVariable = 3;
someObject.someAttribute = myVariable + 1;
```

The equals sign (=) in an assignment statement indicates assignment, not equality. Equality or equivalence in action language are indicated by a double equals sign (==).

Arithmetic The action language has the following simple arithmetic operators, shown here in descending order of precedence:

Operator	Meaning
*	Multiply
/	Divide
%	Modulus (remainder after division)
+	Add
-	Subtract
<< >>	Bitwise left or right shift, respectively.
&	Bitwise and
^	Bitwise complement
	Bitwise or

The << and >> operators can also be used for external C++ streams; they are passed through to the C++ compiler so that you can do things like:

```
extern ostream cout;
cout << "Hello World" << endl;
```

You can use parentheses in expressions to group items together.

```
1 + 2 * 4       // yields 9
(1 + 2) * 4     // yields 12
```

The following section describes all the action language operators.

Operators Unary and binary operators supported on fundamental IDLos types are shown in the following table.

operators	operand type	description
`true` `false` `empty`	boolean	Literals
!	boolean	Not
&&	boolean	Boolean AND Boolean OR
=	boolean, char, enum, long, short, unsigned long, unsigned long long, unsigned short, float, double, string, octet	Assignment
< > <= >=	char, float, double	Numeric comparison
< > <= >=	string	Lexographic comparison
==	boolean, char, enum, long, short, unsigned long, unsigned long long, unsigned short, float, double, string, octet	Equality/equivalence compare. Note: enum operands must be the same type.
+ - * / % << >>	float, double, long, short, unsigned long, unsigned long long, unsigned short	Add, subtract, multiply, divide, modulus, shift left, shift right

operators	operand type	description
+	string	Concatenate
+= -= *= /= %= ^= = &=	These assignment operators perform both an arithmetic operation (or string concatenation, for +=) as in the descriptions above, together with an assignment. You use these operators anywhere you can use the ordinary assignment operator.	
<<= >>=	any	Copy to or from an any. These are the only operators supported for the any data type.

Manipulating Strings

This section is concerned with manipulating string types and implicit type conversions where strings are involved.

Although it represents a simplification of the string implementation, for the sake of the discussion on string manipulations, this and the following sections present strings as having an internal character array, `buffer` , and a `length` attribute; `buffer` grows and shrinks automatically as required; `length` indicates the number of bytes currently in `buffer` ; the internal buffer's index is zero based.

String operators The following operators are defined for strings:

Operator	Description
[]	s[i]:
string, long -> char	buffer[i], for 0<=i<=s.length-1
=	str1 = str2:
string = string -> string	str1', where for 0<=i<=str2.length, str1'[i]=str2[i], str1'.length=str2.length
=	str1 = nonstr:
string = nonstring -> string	str1 = str, where str is a string constructed from nonstr. (See the section called "Constructing a string from a non-string type" on page 68)
+	str1 + str2:
string + string -> string	str, where for 0<=i<=str1.length-1, 0<=j<=str2.length-1, str[i]=str1[i], str[str1.length+j]=str2[j], str.length=str1.length+str2.length
+	str1 + nonstr:

Operator	Description
string + nonstring -> string	str1 + str, where str is a string constructed from nonstr. (See the section called "Constructing a string from a non-string type" on page 68)
+	chrs + str1:
char[] + string -> string	str + str1, where str is a string constructed from chrs (See the section called "Constructing a string from a non-string type" on page 68)
+=	str1 += str2:
string += string -> string	str1 = str1 + str2
==	str1 == str2:
string == string -> boolean	true if str1.length = str2.length and for 0<=i<=str1.length-1, str1[i]=str2[i]
	false otherwise
>, <, >=, <=	Lexicographical comparison.
string (>, <, <=, <=) string -> boolean	Pointers to the internal buffers of the string values str1 and str2 are passed to the standard UNIX function `memcmp` , along with the smallest length value, to determine the whether str1 is lexicographically less than, equal to, or greater than str2.
	str1 > str2:
	true if str1 is greater than str2; otherwise false
	str1 < str2:
	true if str1 is less than str2; otherwise false
	str1 >= str2:
	true if str is greater than or equal to str2; otherwise false
	str1 <= str2:
	true if str1is less than or equal to str2; otherwise false

These operators are only defined for true string types, not string constants. A string constant is handled like a char[] (see the section called "Constructing a string from a non-string type" on page 68)

String operations In addition to the string operators, KIS strings have a number of operations built in. (The string operations for converting strings to other types are handled separately in the section called "Constructing a non-string type from a string" on page 68)

The following table lists and describes some of the most widely used string operations. For a complete list, see Chapter 8.

Operation	Description
string substring(in long start, in long end)	Returns a substring of the string, bounded by start and end, inclusive. For example: `s1 = "01234567890123456789"; s2 = s1.substring(3, 7);` Now s2 contains “34567”. See the following section the section called “sub string errors” on page 67for start and end boundary errors.
void trim()	Returns the string with any leading and trailing whitespace removed.
string toupper()	Returns the string as all uppercase.
string tolower()	Returns the string as all lowercase.
getCString()	Returns the string as a C-style char[]. Useful for printf statements and other places where a C string is required.
length	An attribute (not an operation!) containing the number of characters in the string.

void remove(in long start, in long end)	Removes characters from the string from start to end, inclusive. For example:
	
	`s1 = "01234567890123456789";`
	
	`s1.remove(3, 7);`
	
	Now s1 contains “012890123456789”.

void pad(in long len, in char padchar)	Appends the character padchar to the string to make the string len characters long.
void insertChar(in long pos, in char data)	Inserts the character data into the string at the position pos.
void insertString(in long pos, in string data)	Inserts the string data into the string at position pos.

String operations may be performed on strings you declare in action language, but not on string attributes. You must copy the attribute and manipulate the copy. When you’re done, you can assign the copy back to the string attribute.

sub string errors The function `substring` throws an exception in cases where start > end, start < 0 or end >= length. Failure for any of these three tests will cause the exception, `ExceptionArrayBounds` , to be thrown.

Constructing a string from a non-string type

The KIS string type implicitly constructs a string from boolean, char, and numeric values, as well as string constants, whenever you assign or concatenate one to a string. This section describes how such strings are constructed.

For numeric and char values, the KIS string type relies on the standard UNIX functions `sprintf` and `snprintf` to convert the values to a C-type string, which it writes to the new string's internal buffer.

The following table lists the type and shows the call parameters for converting each type.

Type	Function
char	sprintf(buffer, "%c", value)
double	snprintf(buffer, sizeof (buffer), "%f", value)
float	snprintf(buffer, sizeof (buffer), "%f", value)
long	sprintf(buffer, "%ld", value)
long long	sprintf(buffer, “%llu”, value)
octet	sprintf(buffer, "%d", (SW_INT32)value)
short	sprintf(buffer, "%hd", value)
unsigned short	sprintf(buffer, "%hu", value)
unsigned long	printf(buffer, "%lu", value)
unsigned long long	sprintf(buffer, “%llu”, value)

For details on the UNIX functions, please refer to the corresponding man pages.

An array of char (char[]) is converted to a string by copying the elements of the char array to the string's internal buffer. The conversion uses the standard UNIX function `strlen` to determine the size of the char array. As a result, the conversion only reads up to the first null char.

A string constant is handled like an array of char.

Constructing a non-string type from a string

The string type provides a number of conversion operations that you can invoke on a string to construct a corresponding char, boolean, or numeric value. All of the conversion operations throw the exception `swbuiltin::ExceptionDataError` if the conversion fails.

Constructing boolean values Invoke the `stringToBoolean()` operation on a string value to construct a boolean value from a string. The operation fails if the string is not equal to “true”, “TRUE”, “false”, or “FALSE”. Here is the syntax:

```
boolean stringToBoolean()
```

Constructing char values Invoke the `stringToChar()` operation to construct a char value from string. The operation fails if the length of the string is not equal to 1. Here is the syntax:

```
char stringToChar()
```

Constructing numeric values The operations for constructing numeric types from a string are based on the standard UNIX functions `strtol` , `strtoll` , `strtoul` , `strtod` , and `stroull` . The conversion is performed up to the first non-numeric character.

All but the double and float conversion operations accept an argument of type `NumericBase` . `NumericBase` is an `enum` defined in the string type.

```
enum NumericBase
{
   BaseDerived = 0,
   Base8 = 8,
   Base10 = 10,
   Base16 = 16
};
```

Use the following values to specify the corresponding base: `SWString::BaseDerived` , `SWString::Base8` , `SWString::Base10` , `SWString::Base16` . The default is `SWString::BaseDerived` , which causes the function to determine the base from the string value:

- decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits
- octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only
- hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

The following table lists the numeric conversion operations and UNIX function calls used to generate each numeric type from a string:

Operation	**UNIX Function**
double stringToDouble()	strtod(buffer, &ptr)
float stringToFloat()	strtod(buffer, &ptr) The result is cast to a float.
long stringToLong(in NumericBase base)	strtol(buffer, &ptr, base)
long long stringToLongLong(in NumericBase base)	strtoll(buffer, &ptr, base)
octet stringToOctet(in NumericBase base)	strtoul(buffer, &ptr, base) The result is cast to an octet.
short stringToShort(in NumericBase base)	strtol(buffer, &ptr, base) The result is cast to a short.
unsigned short stringToUShort(in NumericBase base)	strtoul(buffer, &ptr, base) The result is cast to an unsigned short.
unsigned long stringToULong(in NumericBase base)	strtoul(buffer, &ptr, base)

Operation	UNIX Function
unsigned long long stringToULongLong(in NumericBase base)	strtoull(buffer, &ptr,base)

To construct a numeric type from a string invoke the corresponding operation on a string.

Implicit conversion Whenever you assign a string to a numeric type, or use a string in an expression requiring a numeric value, the string attempts to construct a numeric value. It uses the conversion operations listed above to construct the new value.

String manipulation examples

The following code illustrates string manipulations:

```
package stringExamples
{
   [local]
   entity startup
   {
      [initialize]
      void doIt();
      action doIt
      {`
         declare string str;
         declare string even;
         declare string s1;
         declare string s2;
         declare string numStr;
         declare long i;
         declare long l = 0;

         s1 = "1234";
         s2 = 5678;
         str = s1 + s2;
         str += 9;
         for (i=0;i<str.length;i++)
         {
            numStr=str[i];
            l += numStr.stringToLong()*2;
            even += numStr.stringToLong()*2;
            even += '+';
         }
         even += 0x14;
         l += 0x14;
         even += '+';
         even += 0x16;
         l += 0x16;
         printf("1 even: \"%s=%d\"\n", even.getCString(), l);
         declare char   charArray[100];
         sprintf(charArray, "Even numbers: ");
         even = charArray + even + " ";
         printf("2 even: \"%s\"\n", even.getCString());
         even.remove(0, 12);
         printf("3 even: \"%s\"\n", even.getCString());
         even.trim();
         printf("4 even: \"%s\"\n", even.getCString());
         // insert null char
         even.insertChar(7, '\0');
         printf("5 even: \"%s\"\n", even.getCString());
         even = even.substring(0, 6) +
                even.substring(8,even.length-1);
```

```
printf("6 even: \"%s\"\n", even.getCString());
even.replaceChar('+', ' ');
printf("7 even: \"%s\"\n", even.getCString());

declare char    x[100];
x[0] = 0x64;
x[1] = 0x81;
x[2] = 0x80;
x[3] = '\0';
str = x;
printf("1 str: \"%s\" , length: %d\n", str.getCString(),
                          str.length);
str = str.substring(0, 0);
printf("2 str: \"%s\" , length: %d\n", str.getCString(),
                          str.length);
declare octet oct;
oct = str[0];
// oct = 0x64 = 100
str = oct;
// str = "100"
printf("3 str: \"%s\" , length: %d\n", str.getCString(),
                          str.length);
declare long myLong1;
declare long myLong2;
myLong1 = str;
str += "text";
myLong2 = str;
if (myLong1 == myLong2)
{
   printf("Numeric conversion performed up to first
           non-numeric character.\n");
}

sprintf(x, "true or false");
declare string s;
s = "true or false";
declare string boolStr;
declare boolean boolVal;
boolVal = ((x==s) && (s==x));
boolStr = boolVal;
if (boolVal)
{
   printf("1 boolStr: %s\n", boolStr.getCString());
}

if (s.endsWith("false"))
{
   boolStr = s.substring(s.indexOf("false"),
                         s.indexOf("false")+4);
}
boolStr = boolStr.toupper();
printf("2 boolStr: %s\n", boolStr.getCString());
boolVal = boolStr.stringToBoolean();
if (! boolVal)
{
   boolStr = boolVal;
   printf("3 boolStr: %s\n", boolStr.getCString());
}
boolStr.reset();
boolStr.stringNAppend(4, s.getCString());
boolVal = boolStr.stringToBoolean();
if (boolVal)
{
   boolStr = boolVal;
   printf("4 boolStr: %s\n", boolStr.getCString());
}
```

```
        SWBuiltIn::stop(0);

      `};
   };
};
```

Running a component generated from the above code produces the following output:

```
1 even: "2+4+6+8+10+12+14+16+18+20+22=132"
2 even: "Even numbers: 2+4+6+8+10+12+14+16+18+20+22 "
3 even: " 2+4+6+8+10+12+14+16+18+20+22 "
4 even: "2+4+6+8+10+12+14+16+18+20+22"
5 even: "2+4+6+8"
6 even: "2+4+6+8+10+12+14+16+18+20+22"
7 even: "2 4 6 8 10 12 14 16 18 20 22"
1 str: "d\201\200" , length: 3
2 str: "d" , length: 1
3 str: "100" , length: 3
Numeric conversion performed up to first non-numeric character.1 boolStr: true
2 boolStr: FALSE
3 boolStr: false
4 boolStr: true
```

Data types

Action language uses the same fundamental types as IDLos does. You declare a variable using the action language `declare` statement.

Your model's action language can use any built-in type, or any user-defined type in the package where the action language appears. Types in other packages may also be used by qualifying them explicitly, as in:

```
packageName::typeName
```

Use of C-style pointers

The use of C-style pointers in action language is not recommended. However, they can be successfully used in some cases.

The most important restriction on pointers is that you cannot dereference a pointer in an lval. For example, the following action language will fail to parse:

```
char * resultPtr = 'abc';
char SEP = ' ';
*resultPtr++ = SEP; // parser fails here.
```

because the action language parser does not allow the dereferencing of the pointer to `resultPtr` .

Restrict your use of pointers to user-declared data and only in simple rvals. Better still, avoid the use of pointers altogether.

In certain cases, you can effectively dereference in an lval by using an array index as in the following example:

```
      char    a[80];
      char    *p;
```

```
        p = a;

        //
        // *p++ = 'F';
        // *p++ = 'O';
        // *p++ = 'O';
        // *p++ = '\n';
        // *p++ = 0;
        //
        p[0] = 'F';
        p += 1;
        p[0] = 'O';
        p += 1;
        p[0] = 'O';
        p += 1;
        p[0] = '\n';
        p += 1;
        p[0] = 0;

        printf("a = %s\n", a);
```

Control structures

Loops

The action language provides three ways to iterate over a set of statements: `for` , `for...in` , and `while` .

for The `for` statement lets you loop using a range of values. It is identical to the C++ for statement:

```
for( x=0; x<myLimit; x++ )
{
    // do something
}
```

for...in This statement iterates over a set of objects, using a different object each time through the loop:

```
for thisCustomer in setOfCustomers
{
    // do something
}
```

The `for...in` statement may iterate over the instances of an extent or over the object in a relationship:

- extent—if the name of an entity appears after the `in` keyword, then the loop iterates over the extent (all the instances) of that entity
- relationship—if a relationship navigation appears after the `in` keyword, then the loop iterates over the related objects

while This statement loops as long as a given expression remains true. The expression is evaluated at the beginning of each loop:

```
while( money > 0 )
{
```

```
    // spend some money
}
```

Branches

The action language supports `if` , `else` , and `else if` constructs:

```
if( thisCustomer.balance < thisCustomer.creditLimit )
{
    // sell to the customer
}
else if( thisCustomer.alreadyWarned )
{
    // deactivate account
}
else
{
    // warn customer
}
```

Manipulating objects

This section introduces the principal ways that you can manipulate objects. It covers creating and deleting objects, handling object references, accessing the operations and attributes of objects, and using relationships.

Creating objects

To create a new object in action language, first declare a variable of the object's type (see the section called " Object references" on page 63) and then use it with the `create` statement:

```
declare Customer someCustomer;
create someCustomer;
```

This creates a new Customer object in the local shared memory, which the handle `someCustomer` refers to. When the current action finishes, `someCustomer` will no longer exist (see the section called " Scope and lifetime" on page 62) but the new object will remain in shared memory. The object remains even when the handle to it no longer exists. You can locate the object again by selecting from the Customer extent or via any objects that you relate it to (see the section called "Relating and unrelating objects" on page 77).

With initial values You can also create an object with initial values assigned to some or all of its attributes. For example, you can create the customer and provide an initial value:

```
declare Customer someCustomer;
uniqueKey = self.allocateCustId(); // define this somewhere
create someCustomer values (id:uniqueKey);
```

(The identifier `self` never needs to be declared; it always refers to the object in which the action is executing.)

When you create objects that must have unique keys, use the `values` clause to set these keys during creation to avoid creating objects with duplicate keys.

If an entity is marked in IDLos with the `singleton` property, you must use the `create singleton` statement (see the section called "Singletons" on page 75) to create it.

Deleting objects

To remove an object from shared memory, use the delete statement:

```
delete someCustomer;
```

Singletons

If an entity is marked in IDLos with the `singleton` property, you must use the `create singleton` statement to create it:

```
// PortAllocator was defined as a singleton in IDLos
declare PortAllocator thePortAllocator;
create singleton thePortAllocator;
```

This creates a new `PortAllocator` object in shared memory unless one exists already, in which case it simply makes `thePortAllocator` refer to the existing object.

Object references

The preceding paragraphs have used object references in declarations, create statements, and other ways. When you first declare an object, it does not yet refer to anything—it is simply an empty reference:

```
declare Customer myNewestCustomer;
myNewestCustomer.name = "KIS";   // invalid!
```

You need to create the object or assign the handle to an existing object before you can use it:

```
declare Customer myNewestCustomer;
create myNewestCustomer;
myNewestCustomer.name = "KIS";   // this is OK
```

The "empty" keyword Sometimes you need to test whether an object reference is valid or not. For example, a select (see the section called "Selecting objects" on page 78) may not return a valid object:

```
declare Customer aCustomer;
select aCustomer from Customer where (aCustomer.id == 1);
aCustomer.name = "KIS"; // might be invalid!
```

In cases where you might have an invalid object reference, you can check it using the `empty` keyword:

```
declare Customer aCustomer;
select aCustomer from Customer where (aCustomer.id = 1);
if( empty aCustomer )
{
    // whatever you do when there's no customer 1
}
else
```

```
{
    aCustomer.name = "KIS"; // OK
}
```

Operation and signal parameters

When you define an operation (or signal) you can specify formal parameters that the operation takes. The following paragraphs describe the calling conventions and notation for the actual parameters that you supply when you call the operation from action language.

Calling conventions Actual parameters that you pass in when you call an operation or signal are passed either by reference or by value:

- objects are always passed by reference
- other types (fundamental, complex, etc.) are always passed by value

Positional and named parameters The action language lets you call operations and signals using either positional or named parameters. If you use any named parameters, you must name all parameters — you cannot use named and positional parameters in the same call.

For example, given the IDLos:

```
entity CallingOpsAndSignals
{
    signal aSignal( in long inL, in boolean inB);
};
```

you can invoke the signal in action language using either named or positional parameters:

```
declare long        aLong = 1;
declare boolean     aBoolean = false;

//
// Call aSignal using positional parameters
//
self.aSignal(aLong, aBoolean);

//
// Call aSignal using named parameters. Notice that
// the parameter order was reversed.
//
self.aSignal(inB:aBoolean, inL:aLong);
```

Accessing operations and attributes

As demonstrated in previous examples, you access the attributes of an object using the “dot” operator. This is also how you access its operations:

```
if( thisCustomer.creditLimit > requestedAmount )
{
    thisCustomer.sendApprovalLetter();
}
```

You can’t access nested members by chaining together multiple “dot” operators:

```
declare short         since;
since = thisCustomer.profile.customerSince; // illegal
```

Instead, you need to use an intermediate variable:

```
declare CustProfile   thisProfile;
declare short         since;
thisProfile = thisCustomer.profile;
since = thisProfile.customerSince;
```

Handling relationships

A relationship between entities in an IDLos model specifies the objects that may be related to each other. Relating specific objects to other objects is something that you do at run time using the role name:

Relating and unrelating objects A relationship between entities in an IDLos model specifies the objects that may be related to each other. Relating specific objects to other objects is something that you do at run time using the role name:

```
declare customers::Customer thisCust;
declare billing::Invoice thisInvoice;

//
// Select customer, create invoice, and relate the two
//
select thisCust from customers::Customer where (thisCust.id=123);
create thisInvoice;
relate thisCust billing::isBilledBy thisInvoice;
```

This leaves the two objects related in the shared memory, so that you can always find one given the other. For example, you can process all the invoices for a customer:

```
for oneInvoice in thisCust->billing::Invoice[billing::isBilledBy]
{
    // process the invoice
};
```

When you no longer want two objects to be related, use the `unrelate` statement:

```
unrelate thisCust sales::belongsTo thisSalesRep;
```

When you delete an object that is related to another object, the Application Server does an automatic unrelate for you. You are not left with a dangling relationship from the surviving object. If you are using unrelate triggers, note the specific behavior of the triggers for this implicit unrelate, described in "Role trigger" on page 50.

Navigation You can "navigate" (traverse) across multiple relationships to find one object that is related to another via one or more intermediate objects. For example, suppose for each overdue invoice we want to notify the manager of the customer's sales representative:

```
declare
{
    billing::Invoice theInv;
    sales::SalesManager theMgr;
}
```

```
for theInv in billing::Invoice
{
    if( theInv.isLate )
    {
        theMgr = Invoice
                 ->Customer[billing::bills]
                 ->SalesRep[sales::belongsTo]
                 ->SalesManager[sales::worksFor];
        theMgr.notify();
    }
}
```

Selecting objects You use the select statement to select a single object from a relationship, extent, or singleton. You always include a `where` clause:

```
select aCustomer from customers::Customers where (aCustomer.id=123);
```

except when you're selecting a singleton or from a "one" end of a relationship:

```
select aCustomer from thisInvoice->Customer[billing::bills];
```

Using Keys Keys cause KIS to maintain an index in shared memory that can be used to optimize access to objects. The correct use of keys is critical to building performant applications.

Non-Unique Queries

```
entity E
{
        attribute long  unique_id;
        attribute long  index_id;
        attribute long  unkeyed_id;

        //
        // This is the default.
        //
        [ unique = true ]
        key Unique { unique_id };

        //
        // This specifies that the key is non-unique, you can have
        // multiple instances in shared memory with the same key
        // value.
        //
        [ unique = false ]
        key Duplicate { index_id };
};
```

To access objects using the Duplicate key, a for statement must be used.

```
for e in E using Duplicate where (e.index_id == 42)
{
        count += 1;
}
```

Notice the using clause in the above for statement. The using clause indicates that the Duplicate key should be used to perform the query. The action language auditor will generate a warning if the using clause is missing, because without the Duplicate key, this action language statement will perform a complete extent iteration.

Attempting to use a select statement with a non-unique key will fail action language auditing. For example:

```
//
//  This is an illegal select since multiple objects could be
//  returned
//
select e from E using Duplicate where (e.index_id == 42);
```

Range Query

A range query is specified using a where clause in a for loop that defines a range of values for an ordered key. A range is specified using >, <, >=, and <= operators. Multiple ranges can be specified, and multiple keys can be used to aggregate a result set. Transaction locking can be specified for iterations over a range query.

The range operators built into action language are used for range queries. This implies that the comparisons done to determine range queries will give the same results as the comparisons done on standard action language types. String comparisons use a built in lexical comparison, treating each character value as an octet.

```
entity SomeObject
{
        attribute swbuiltin::date createDate;
        attribute long id;
        attribute string lastName;
        attribute string firstName;

        [ unique=false, ordered=true ]
        key DateKey { createDate };

        [ unique=true, ordered=true ]
        key IdKey { id };

        [ unique=false, ordered=true ]
        key NameKey { lastName, firstName };
};
```

All examples below use the entity definition above.

```
//
// Simple date compare, uses builtin date comparision operators
//
for obj in SomeObject using DateKey where (obj.createDate >= someDate) writelock
{
}


//
// Simple string compare, uses builtin string lexical order.
//
for obj in SomeObject using NameKey where (obj.lastName >= lastName) nolock
{
}


//
// Range Name compare, with firstName restricting result set
//
for obj in SomeObject using NameKey where
        ((obj.lastName >= lastName && obj.firstName == “Dan”) readlock
{
}

//
// Multikey or'd. The result set will be a unique set of instances that match
// either range.
```

```
//
for obj in SomeObject using DateKey, IdKey where
        ((obj.createDate >= startDate && obj.createDate <= endDate) ||
        (obj.id > 100)) writelock
{
}

//
// Multikey and'd. The result set will be a unique set of instances that match
// both ranges.
//
for obj in SomeObject using DateKey, IdKey where
        ((obj.createDate >= startDate && obj.createDate <= endDate) &&
        (obj.id > 100)) nolock
{
}
```

The examples below will fail action language audit.

```
//
// Only firstName used, not allowed
//
for obj in SomeObject using NameKey where (obj.firstName >= someName)
{
}

//
// Also fails if firstName used in a OR expression.
//
for obj in SomeObject using NameKey where
        (obj.lastName >= lastName || obj.firstName <= firstName)
{
}
```

Ordering Query Results

When iterating over an ordered key, the order of the results can be specified. By default, the order of a range query is not guaranteed; the model must specify an order by clause to insure the result set is ordered.

The examples below show how results sets can be ordered. The examples use the entity definition above.

```
//
// Result set ordered by last name
//
for obj in SomeObject using NameKey where
        (obj.lastName >= lastName) order by lastName ascending nolock
{
}

//
// Result set ordered by creation date
//
for obj in SomeObject using DateKey where
        (obj.createDate >= someDate) order by createDate descending writelock
{
}

//
// Multi-order. Result sets are returned in order specified:
//
// first order sort: lastName
// for duplicate lastName, second order sort: firstName
//
```

```
for obj in SomeObject using NameKey where
        (obj.lastName >= lastName) order by
        lastName decending, firstName ascending readlock
{
}
```

The examples below will fail action language audit.

```
//
// Must use same attributes in using key in order by clause
//
for obj in SomeObject using NameKey where
        (obj.lastName >= lastName) order by createDate descending
{
}

//
// Must use first attribute in NameKey in order by clause
//
for obj in SomeObject using DateKey where
        (obj.createDate >= someDate) order by firstName descending
{
}

//
// Ordering not supported for multi-key ranges
//
for obj in SomeObject using DateKey, IdKey where
        ((obj.createDate >= startDate && obj.createDate <= endDate) &&
        (obj.id > 100)) order by createDate ascending
{
}
```

4

Building KIS components

When you build your model into a component, everything needed to deploy the model on a Kabira Infrastructure Server (KIS) node is wrapped into a single file.

This chapter describes the KIS tools and commands you use to generate a KIS component. To build a component from your model you:

1. Design an implementation

2. Build the model into a deployable component

A component specification is used to build an implementation of your model and is introduced in this first section of the chapter:

This chapter is a guide to developing a component specification. It is presented as follows:

The final section of the chapter describes how you build a deployable component from your component specification:

• the section called “Building the component” on page 89

The KIS component

A KIS component is an executable model that can be deployed and reused. To reuse a component, you import the component providing the service into a new component.

You turn your model into a component using a component specification.

What is a component specification?

A model is an abstract definition of your application logic, with little or no consideration for implementation. Your component specification, by contrast, specifies how you want the model implemented. A component specfication defines:

- model source files to include
- model elements to implement in the component
- service adapters to implement for model elements
- dependencies with other components
- compile time options

You use a component specification to build applications from models. It describes how to turn your model into a deployable KIS component.

Defining a component specification

To define a component specification is a three step process.

1. Populate the specification. Select the elements from your model for each part of the implementation (for example: package A for component A, interface B for adapter C).
2. Add properties. Select and add the properties to the model elements you are implementing.

Figure 4.1 shows a conceptual view of this process

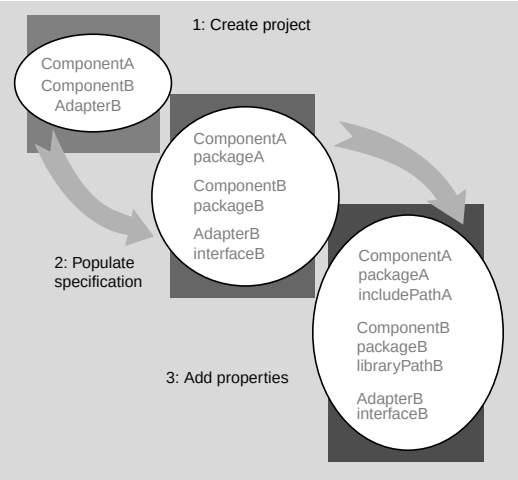

Figure 4.1. The three general stages in defining a project

Although Figure 4.1 shows this as three discrete stages you can mix the individual steps in any way that is convenient. The order is not important.

The individual steps by which you define a project are described in greater detail in the following sections.

Properties

You can define properties, such as buildPath and classPath, globally. If you set properties at the global level they will be defined for the entire component specification. For a complete list of global properties, see “Properties” on page 477.

To set a global property, define the property outside the scope of all the components in the component specification file. For example:

```
// project property
buildPath = “some/build/path”;
component Component1
{
};
component Component2
{
};
```

Working with model sources

The core of the project is your model sources. The component specification is used to build a deployable or reusable implementation of this model.

You identify the sources for a component with source keyword. These files will be loaded into the Design Center server prior to a build. (On the Design Center Server, see Chapter 14) Source files must be listed in the correct order of dependency. If you use a source statement in the component specification rather than #include directives in IDLos files, the Design Center server can optimise reloading.

```
component MyComponent
{
   source /path/to/myfile.soc
};
```

Defining a component

A component specification will contain all the information necessary to build a deployable and reusable component from your model.

Adding a new component

You start the textual component specification with the component keyword. This starts the definition of a component block. For example:

```
component TrafficComponent
{
    // a component specification for the Traffic applicaton
};
```

A component implements model packages. It may contain adapters, or model elements; these may be grouped.

Component properties

There are a number of properties that you can set for a component that control how the compilers generate code. More than one property may be set for a component. For a complete list of properties available at the component level, see “Properties” on page 477.

To set a component property, define the property inside the component block in the component specification file.

```
component Component1
{
    includePath = “some/include/path”;
}
```

Many of these properties can also be set globally so that they apply to all the components contained in the build. For more information refer to the section called “” on page 85.

Putting packages in a component

You add complete packages to a component. It is not possible to add some entities in a package to one component, and other entities in that package to another component.

The keyword package signifies an element type. A package can belong to a component, an adapter, or a group.

```
component MyComponent
{
    package MyPackage;
}
```

You cannot partition a package across components, but you can implement a package in more than one component. This constraint should be taken into consideration at the modeling stage.

Importing another component

You can use a component as a service in another component. To do this, you import one component—the one providing the service—into the other. You will need to create a client model to use that new service. For example, the client model may instantiate and use interfaces in the service component. Figure 4.2 illustrates this concept.

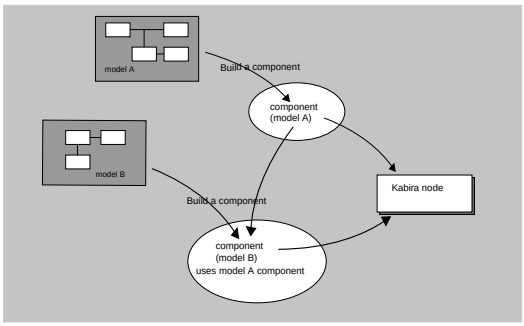

Figure 4.2. Two models used as client and server components

When a component A depends upon another component B, you declare A's dependency upon B with the import statement. In the example below, the default include path is augmented with the includePath property for all components referenced in component A.

```
component A
{
    importPath = /some/path;
    import B;
};
```

Adding adapters

You add adapters to a component to implement selected entities or interfaces in a database or service. For example, if you want selected entities in a component to be implemented in Oracle, you would add an oracle adapter to the component. When you build your application, the component will include code to maintain and synchronize the database and shared memory copies of its objects.

You add an adapter to a component with the adapter keyword followed by the name of the adapter. The example below shows how component A uses the CORBA adapter swiona to implement the interface ::MyPackage::MyInterface:

```
component A
{
    adapter CORBA
    {
        interface ::MyPackage::MyInterface;
    }
};
```

Adapter properties

There are a few properties that you can set for adapters. You add and edit adapter properties like you add and edit global and component properties. Each type of adapter has different properties; for these properties and the values they can take, refer to the documentation for each individual adapter.

Adding model elements to adapters

When you add an adapter to a component you also need to designate which entities or interfaces, depending on the adapter type, are used with that adapter.

For instance, you connect a modeled entity to an external implementation, such as a database, using a database adapter. You connect a model interface to an external protocol such as CORBA or SNMP using a protocol adapter. Refer to the specific adapter in each service adapter for more information on what can be implemented in that adapter.

Adding an element

The model element types are: package, module, interface, entity, relationship, operation, signal, attribute, role and key.

```
component MyComponent
{
    adapter CORBA
    {
        interface ::MyPackage::MyInterface;
    };
    adapter Oracle
    {
        entity ::Mypackage::MyEntity
        {
            updateString = “Some SQL String”;
        };
    };
};
```

Model element properties

Implementation adapters let you set properties on an entity to control how those objects are cached, and to support legacy databases. Refer to the documentation for Database Adapters to learn how to use these properties.

Some protocol adapters allow you to set properties on the interface. Refer to the documentation for Database Adapters to learn how to use these properties.

You add and edit model element properties within the element block like you add and edit global and component properties. Refer to the section called “” on page 85 for step by step instructions.

Putting relationships and roles into adapters

Refer to the documentation on individual adapters for more about relationships and roles in adapters.

Putting attributes into adapters

You add attributes to an adapter to override the settings for that attribute in the interface or entity. For example, a read-write attribute in an interface can be made read-only for use with an adapter.

Setting attribute properties Some adapters let you customize attributes to override the default definition for the attribute in its enclosing entity or interface.

You add and edit attribute properties within the attribute block like you add and edit project and properties.

Building the component

This section describes how you build a deployable component. Before you can build the component, you must have selected model sources and completely defined your component specification.

Starting a Build

The utility swbuild allows you to build KIS applications from the command line. It requires the model and the component specification files and is as follows:

```
swbuild [options] mySpecFileName
```

By default, swbuild performs both an audit and a build. The [] define options to perform an audit only, use the "-a" option. The optional "-o <macro>=<value>" assigns a value to a macro used in the build (see "Macros" on page 474.) The "spec file" is the Component specification file. If none is specified, standard input is used.

option	description
a	performs an audit
d	Enable debug trace
l	List installed components
g	List design center configuration
r	Generate raw messages with no formatting
m msg-prefix	Prefix msg-perfix to all error messages.
o macro=value	assigns a value to a macro used in the build
p dcpath	Alternate path to Design Center Server. By default swbuild looks for the Design Center Server in the current directory. (On the Design Center Server, see Chapter 14)
P port	Design center port. Default is random
h host	Design center host. Default is localhost.
t timeout	Coordinator timeout in seconds. Default is zero (no timeout)
u user	The user name to use.
w password	User password
N node	Design center node name
c comopnent	Display SDL file for component.

option	description
s searchpath	Set component search path

What is auditing?

Before the build begins the Design Center server will run an audit.This is automatically part of the build process. You may also run a seperate audit before you build.

Auditing your model checks for many different kinds of errors that can be expressed in valid modeling syntax. The audit that takes place before a build includes not only model syntax , but action language too;

This also performs adapter-specific checks for any adapters in the build. Some of the audit checks are:

- Are all element names unique within their scopes?
- Is the model free of inheritance “loops”?
- Are all entity and interface definitions complete?
- Are all modules contained in packages?
- Are all relationships contained entirely within single packages?
- Do relationships have at most one role per direction?
- Do state transitions use signals and states that are defined for the entity?
- Do interfaces expose only what exists?
- Do operations, attributes, parameters, and data types used in action language agree with their IDLos definitions?
- Are relationships traversals in action language possible in the IDLos model?
- Are properties such as key and singleton observed in action language?

What you get after you build

After a successful build, you get a set of output files. By default, these files, and some subdirectories, are created in the directory running the Design Center server. You can alter this location by specifying a directory using the buildPath property. See the section called “” on page 85 or the section called “Component properties” on page 86.

Although the build generates many files, there is only one that you need to be concerned with. That file is the component archive, named componentname.kab (where componentname is the name you gave your component in tthe component block in the build specification file). This is the archive file that you deploy on a KIS node. See Deploying and Managing KIS Applications for more about deploying components.

5

Developing secure KIS applications

This chapter describes the Kabira Secure Services Layer (KSSL), which enables the development and deployment of secure KIS applications. The chapter contains the following principal sections:

In addition to the information in this chapter, see the KSSL online documentation shipped with the KIS core release.

Introduction

This chapter describes the Kabira Security Services Layer (KSSL). KSSL provides a set of services which enable the development and deployment of secure applications built on the KIS platform. The services provided by KSSL are as follows:

- authentication services
- access control services
- security audit services
- secure network communications
- cryptography services

KSSL provides access to multiple industry-standard security mechanisms via a generic modeled interface, allowing user applications to develop secure applications which can be deployed in environments which utilize multiple security technologies.

Taken together, the services provided by KSSL represent a next-generation platform for development and deployment of secure, distributed applications. In this regard, KSSL serves as a key differentiator for the KIS product set.

KSSL encompasses several distinct functional components which provide security services to applications deployed on KIS, and the KIS runtime environment itself. This document specifies the functional behavior of the various KSSL service components. This document also describes security enhancements to existing KIS product components.

Overview of security concepts

Security considerations are a key part of the design, implementation and management of information systems. Ensuring that access to resources is allowed only to individuals who are authorized to do so, and that such access occurs only through private means are essential aspects of any secure system.

The term security is used to describe a wide range of services, any of which may be independently implemented. These services are as follows:

Authentication The ability to verify the identity of a given principal. The term principal refers to a user or agent on whose behalf work is to be done in the system. The process of authentication incorporates verifying that a principal's credentials prove the identity of the given principal.

Access control The ability to define access rules for a given resource in the system. Such rules are typically specified by granting or revoking a given privilege to a given principal or role. A role describes a user type, such as "customer", "administrator", etc.

Confidentiality The ability to carry out a dialog, typically across a network connection, such that the communication occurs between two authenticated principals, and that it cannot be understood by an eavesdropper.

Cryptography Technology which converts messages to/from an illegible format, is used to achieve this purpose.

In addition to the services listed above, confidentiality further requires two additional services:

Replay detection This mechanism ensures that messages are not intercepted and replayed by an eavesdropper.

Non-repudiation These services ensure that a sender may not deny that a given message was sent or a receiver cannot deny that a message was received.

These additional services are typically provided by means of message digest mechanisms, which generate unique signatures for a given message.

Delegation The ability of a system to carry out tasks on behalf of a given principal. A secure system can allow authenticated principals to request work to be done, and have that work occur in other parts of the system with assurance that the authentication is still in effect.

These functional areas are addressed to varying degrees by existing technologies. Confidentiality in network communications has been formalized in widely-adopted standard mechanisms such as

Secure Sockets Layer (SSL), Transport Layer Security (TLS), and the emerging IPv6 protocol. The areas of authentication, credential management, and access control, however, are still rife with multiple competing and incompatible mechanisms, formats, and implementations. Enterprise environments face the challenging task of consolidating these disparate systems into a manageable and interoperable whole.

Authentication systems

Authentication systems, and the corresponding credential formats, have gone through evolutionary changes. The major technologies, in roughly chronological order of development, are described as follows:

Simple text user name and password This mechanism was implemented in the first multi-user operating systems, and relies on a secure database of user name/password pairs which are maintained and verified at the entry point to the protected resource. UNIX login accounts are a good example.

Token-based authentication This mechanism incorporates an authentication service running on a network, which provides authenticated principals with a token which is presented to secure resources. These resources utilize the given token to verify the identity of the given principal. Access to the authentication service is secured through the use of a shared secret between each user of the system and the authentication service itself. Examples of this mechanism would be KerberosV4 and DCE.

Public-key-based authentication This mechanism utilizes public/private key pairs to identify principals and resources in the system. Briefly stated, public key authentication relies on the fact that a message decrypted with a given principal's public key would be meaningful only if that message was encrypted with that principal's private key. Public keys are ideally published in a widely accessible repository, such as a directory. Such keys are typically packaged in a certificate, which incorporates various attributes which describe the principal, including the public key. Private keys, of course, are not published, and are presumed to be stored in a manner that may only be accessed by the principal owner of that key. Authentication occurs by verifying the validity of a given certificate. That verification involves access to a certificate authority (CA) whose public key was used to generate a digital signature for the principal's credentials. A prime example of public key based authentication X.509 systems.

KIS security architecture

The design of the KIS security architecture is guided by the following objectives:

- Provide for secure operational access to KIS nodes.
- Enable development and deployment of secure distributed applications built on KIS.
- Leverage model-level abstraction strengths of IDLos by providing access to multiple security implementations via a generic modeled interface.
- Allow users to add security functionality to existing KIS applications without requiring changes to the application code itself.
- Provide security functionality with minimal performance impact.
- Exploit market opportunities in management of security services in enterprise environments.

These objectives are discussed in subsequent sections.

Secure node administration

KSSL provides support for securing administrative access to KIS nodes. All operational access to a node is secured using standard KSSL mechanisms, providing for seamless integration between node and application-level security policy.

Secure distributed applications

KSSL provides a set of security services which enable users to develop and deploy secure distributed applications. The services provided by KSSL are:

- secure network communication services
- cryptography services
- credential services
- authentication services
- access control services
- security audit services

These services are provided in distinct components, to allow users to utilize only the functionality needed by a given application. The following sections describe each of these functional areas in more detail.

Secure network communication services

Distributed applications require access to secure network communication services. KSSL provides these services via a modeled interface, based on the OpenSSL adapter.

The KSSL transport services component provides support for client and server authentication, data integrity and confidentiality across TCP/IP network connections. The component is integrated with other KSSL components which provide model-level access to credential and authentication services. This integration allows applications to represent all security-based resources used by their application as model elements.

Cryptography services

KSSL provides model-level access to cryptography services, which allow applications to ensure confidentiality of user data at the model-level. The services provided include support for a rich set of cipher suites, key generation and key agreement, and message authentication code generation.

Credential services

KSSL provides model-level representations of principals - users or agents on whose behalf the system will take actions - and credentials which are associated with a given principal. KSSL provides transparent mapping of these model elements to implementation-specific credential data for a wide variety of security system implementations.

Applications use these model-level representations of principals and credentials as input to the various security services provided by KSSL.

KSSL further allows the representation of a principal as a composite identity, having multiple credentials which map to arbitrary implementation-specific credentials. For example, an KSSL principal may contain multiple X.509 credentials and/or username/password credentials, all of which may be examined by the system during authentication of the given principal.

Authentication services

KSSL provides model-level representations of generic authentication services which are used to validate the credentials associated with a given principal. The KSSL authentication service provides access to multiple technology-specific authentication services through a generic interface. That interface includes entry points which allow user applications to initiate and terminate a secure session within the KIS runtime environment.

Access control services

KSSL provides model-level representations of access control list (ACL) constructs, which may be used to define fine-grained rules of access to a given resource. Access control rules may be defined for specific roles, which are associated with principals. The KSSL access control service allows users to specify access control rules for any modeled interface type, and specifically for any attributes and operations in the given type. KSSL further provides the ability to define access control for specific instances of a given modeled interface type, or all instances of the type.

Definition of security policy for a given application via the access control service is dynamic, and may occur at runtime via explicit calls into the service from the application, and/or an administrative console.

Security audit services

KSSL provides a model-level audit service which allows users to request a wide range of security audit services. These services include:

- audit trail for a principal
- audit trail for a trusted resource
- access control activity
- access control violations

The KSSL audit service provides access to activity reports within the model at run-time, and also provides automated logging to disk files.

KSSL service specification

The KIS security architecture consists of several distinct functional components which together provide end-to-end security services to KIS applications.

The KIS security architecture and its components are illustrated in Figure 5.1.

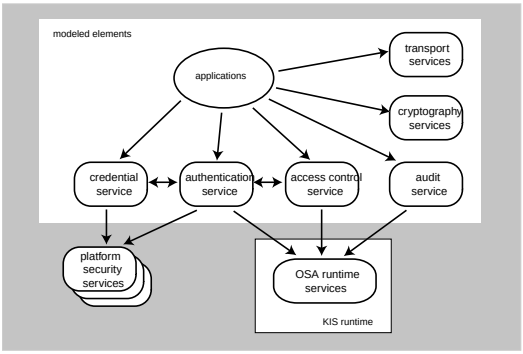

Figure 5.1. KSSL service components

The following sections describe each service, its public interface, its interaction with other KSSL components, and external technologies. The full IDLos specification for all kssl services may be found in the idlosdoc-generated documentation for the KSSL component.

KSSL basics

KSSL relies on a few fundamental concepts, which are described in the following table:

Principal	A logical representation of a user of the system.
Credential	A data value which provides proof-of-identity for a principal.
Role	A label describing an access privilege, as applied to a protected resource, and granted to a principal.
Access control rule	A directive which grants or revokes access privileges to a given resource, to certain roles.
Security policy	A set of directives which specify the set of known principals, their associated credentials and roles, access control rules, and other policy directives.

KSSL provides the ability to specify security policy via the following mechanisms:

- deployment specifications
- KSSL API

These two mechanisms are complementary: security policy defined in a deployment specification may be modified or appended by KSSL API calls from an application component. Security policy directives specified via deployment specifications are applied to a node internally via the KSSL API.

Configuring security policy

Configuration of security policy consists of the following logical operations:

- Configuration of authentication mechanisms.
- Defining the set of known principals.
- Defining the set of roles for each principal.

- Defining the credentials for each principal.
- Defining the set of access control rules in effect.
- Defining Trusted Hosts, if any.
- Defining Security Audit policy, if any.

Each of these logical operations may be accomplished via security policy directives in a deployment specification, or via the KSSL API. The following sections describe each of these operations in detail.

Configuration of authentication mechanisms

KSSL supports multiple authentication mechanisms. The primary mechanism, text credentials, is enabled by default and needs no additional configuration. Additional mechanisms support configuration as outlined in the following sections.

Configuring X509v3 authentication The X509v3 authentication mechanism supports public key-based X509v3 certificate credentials for principal authentication. The mechanism allows KSSL to utilize external Public Key Infrastructure (PKI) services to validate X509v3 certificates. The essential PKI elements involved are outlined in the following table:

Principal	An entity which possesses an X509v3 certificate, which was issued by a Registration Authority (RA) known to the PKI.
Certificate authority	A PKI service which provides for the issuing, validation and revocation of X509v3 certificates.
OCSP responder	A network service provided by a CA to allow dynamic querying of the status of X509v3 certificate.
OCSP client	A client-side API used to request validation status for X509v3 certificates.

KSSL utilizes an OCSP client to request validation status for X509v3 certificates presented as part of principal authentication. The following configuration options, specified via deployment specifications on the node where KSSL security services are to be used, enable this functionality:

```
deploy mynode
{
    config Security
    {
        config AuthenticationMechanisms
        {
            config X509v3
            {
                enabled=<boolean>;
                issuerCertificate=<path>;
                ocspResponderURL=<url>;
                ocspResponderHost=<string>;
                ocspResponderPort=<number>;
                CACertificate=<path>;
            };
        };
    };
};
```

The following table describes each of these configuration options.

enabled	Whether X509v3 authentication is enabled on the given node. The default is "true".
issuerCertificate	The certificate of a CA which issued X509v3 certificates which may be presented to KSSL as principal credentials. This option may be specified multiple times. Any X509v3 certificate presented to KSSL must have had its issuerCertificate configured via this option. The file pointed to by this option must be in PEM format.
ocspResponderURL	The URL to the OCSP Responder to be contacted for certificate validation. This option is ignored if ocspResponderHost/ocspResponderPort are specified.
ocspResponderHost	The host on which the OCSP Responder is running. This option, in combination with ocspResponderPort, may be used instead of ocspResonderURL to specify OCSP Responder location.
ocspResponderPort	The port (on ocspResponderHost) on which to contact the OCSP Responder.
CACertificate	The X509v3 credential(s) of trusted Certificate Authorities. These credentials are used to verify the signature of OCSP responses. This option may be specified multiple times. The file pointed to by this option must be in PEM format.

Defining principals

A new principal may be defined either via a deployment specification, or via the kssl::Context interface. The deployment specification syntax for defining a new principal, including all supported options, is shown in the following example:

```
deploy mynode
{
    config Security
    {
        config Principals
        {
            config Fred
            {
                textCredential=dino;
                roles=Bowler Builder;
                deferredCredential=false;
                credentialExpirationPeriod=0;
                allowEmptyCredential=false;
                trustedHostUser=false;

                // the following options apply to
                // X509v3 authentication only
                x509v3Credential=<path>;
                issuerCertificate=<path>;
            };
        };
    };
};
```

The following table explains each of the configuration options above.

textCredential	The text credential for this principal.
roles	The set of roles defined for this principal, where the given value is of the form: "<role 1> <role 2> ...<role N>"
deferredCredential	If set to true, this option avoids the need to define a textCredential for the given principal, and requests that the textCredential provided on the initial authentication of the principal be accepted and stored as the valid textCredential for that principal. The default value for this option is false.
credentialExpirationPeriod	A numeric value which defines the number of days (24 hour periods) that the credentials for this principal are valid. Once the period as expired, authentication will fail until the credentials are reset. The default value for this option is 0, which disables credential expiration for this principal.
allowEmptyCredential	If set to true, this option allows the textCredential value to be an empty string. The default value for this option is true.
trustedHostUser	If set to true, this option causes authentication from non-trusted hosts to automatically fail. The default value for this option is false.
x509v3Credential	The X509v3 certificate credential for this principal. If this option is present, the certificate will be used to authenticate the principal, according to the configuration options for X509v3 authentication. The file pointed to by this option must be in PEM format.
issuerCertificate	The certificate of the CA which issued the certificate specified via the x509v3Credential option. The file pointed to by this option must be in PEM format.

None of the above options is required if the default value is appropriate.

KSSL supports updates to existing principals. A deployment specification containing a principal definition for an existing principal will be applied as an update to that principal. Only the options specified in the update will be applied to the principal.

X509v3 configuration options for a principal are not required, and may alternatively be provided as part of an authentication request.

Text credential generation

KSSL supports generation of random textCredential values for principals. KSSL automatically generates a textCredential value if a principal definition in a loaded deployment specification lacks the textCredential option.

The Context interface

The kssl::Context interface provides operational control over the security mechanisms used to authenticate principals of the system, and to manage the set of roles associated will principals in the system. This facility is used by secure service applications, and is typically unused by secure client applications.

The following table describes the operations in the Context interface.

enableMechanism()	Enable authentication via the given mechanism. A Properties parameter allows the application to specify mechanism-specific operational parameters to be used during mechanism initialization and authentication activity.
disableMechanism()	Disables use of the given mechanism. Once this call returns successfully, KSSL will ignore any credentials of the given type.
addPrincipal()	Define a new principal in the system. This operation also associates a set of role names with the given principal.
removePrincipal()	Remove the given principal from the system.
getPrincipals()	Retrieve a set of principal names listing the principals defined in the system.
getActivePrincipal()	Retrieve a reference to the principal instance for the currently authenticated principal.
grantRoles()	Grant a set of roles to a previously-de fined principal.
revokeRoles()	Revoke a set of roles from a previously defined principal.
getRoles()	Retrieve the set of roles defined for the given principal.
setAuthenticator()	This operation installs an AuthenticationNotifier object which will be used to carry out application-specific authentication for this mechanism. This facility is further described below.
getCredentials()	Retrieve the set of defined credentials for the given principal.
addTrustedHost()	Define a trusted host for the system.
removeTrustedHost()	Remove a trusted host from the system.

The kssl::Context type is implemented internally by KSSL as a singleton object. Applications acquire an object reference to this object via a "create ... singleton" directive.

The credential service

The KSSL credential service provides a generic credential specification facility, which allows applications to represent credential data and security role values as model-level types.

The kssl::Credential Interface The KSSL credential interface type serves as a container for mechanism-specific credential data. The type has the following attributes:

type	A kssl::SecMechanismType enum value that defines the mechanism that this credential pertains to.
data	A binary representation of the credential data itself.
props	A kssl::Properties value which defines any ancillary information related to this credential.

Lifecycle of Credential instances Applications create instances of the Credential interfaces, populate them, and associate them with principal objects. The Credential instances are required to remain in the system as long as the associated principal object also remains in the system.

The authentication service

The KSSL Authentication Service provides a generic facility for representation of principals (users) in the system, and for authenticating those principals. The service also provides a facility which allows secure services to configure the set of security mechanisms to be used by the service, and to manage the set of roles associated with a given principal.

Authentication consists of validating the credentials associated with the given principal. The validation is done via a mechanism-specific process which is transparent to the calling application.

Trusted-host-based authentication KSSL supports a Trusted Host facility, which allows for expedited authentication of principals when the authentication request originates from a previously-specified Trusted Host. Authentication from a Trusted Host passes without consideration for credentials. That is, if the authentication request originates from a trusted host, KSSL trusts the host-based authentication mechanism (e.g. UNIX login) to have verified the identity of the given principal.

A Trusted Host may be specified via a deployment specification directive, or via the kssl::Context interface. The deployment specification syntax is as follows:

```
deploy myNode
{
    config Security
    {
        config Hosts
        {
            config myHost
            {
                trusted=true;
            };
        };
    };
};
```

In the above example, the host “myHost” is set to be a KSSL Trusted Host.

External authentication—the AuthenticationNotifier interface KSSL allows applications to carry out authentication using application-specific criteria. To utilize this facility, an application must define and implement an interface which is derived from the kssl::AuthenticationNotifier abstract interface. The application must create an instance of the derived interface and install it via the Context::setAuthenticator() operation described above.

The AuthenticationNotifier abstract interface contains the following operation:

authenticate() This operation is invoked by KSSL during authentication for a principal who owns a credential pertaining to a security mechanism for which this notifier was installed. The user-provided implementation of this operation carries out application-specific authentication and indicates whether authentication for the given principal succeeded or failed via its return value.

Once installed, the AuthenticationNotifier instance must remain in existence as long as the corresponding security mechanism is enabled.

The Principal interface Instances of kssl::Principal are used to represent users, or active agents in the system. The kssl::Principal interface contains the following attributes:

name A principalName value that uniquely identifies a user of the system.

The following table describes the operations in the kssl::Principal interface.

clearCredentials()	Disassociates the given principal from all related credentials.
authenticate()	Verify the identity of this principal via the security mechanisms for the associated credentials, and the set of roles previously granted to this principal. This operation raises an AuthenticationFailure exception if the authentication fails.
resetCredential()	Reset an expired credential for the given principal.
endSession()	Terminates the authenticated session initiated via a previous successful call to authenticate().

KSSL requires that Principal objects remain in the system for the duration of any authenticated session for that principal.

Associating principals and credentials

KSSL defines the following relationship between the Principal and Credential interface types:

```
//
// Relationship PrincipalCredentials - allows
// users to associate credentials with a given
// Principal object.
//
relationship PrincipalCredentials
{
    role kssl::Principal hasCredentials 0..* kssl::Credential;
};
```

A principal may have multiple associated credentials. During authentication, KSSL will examine each credential, in the order that they were related to the principal, and carry out authentication with the credential's security mechanism. KSSL attempts to authenticate the principal with all associated credentials, and fails authentication only when authentication has failed for all associated credentials.

While principals may be associated with multiple credentials, those credentials must be for distinct security mechanisms. It is an error to attempt to associate more than one credential for a given se-

curity mechanism with a given principal. KSSL will raise a system exception and abort the current transaction when this error condition occurs.

Chained authentication The KSSL Authentication Service supports the ability to chain multiple security mechanisms when attempting to authenticate a principal. Chaining provides a facility for authenticating a given principal using a set of credentials, as defined for that principal. Each credential is used, in the order specified, and the corresponding mechanism is applied to authenticate the principal.

This facility comes into play when authentication using the X509v3 mechanism is used. A principal which has been specified as having an X509v3 certificate (during principal definition) must first prove their identity to KSSL using a text credential. Once that step in the authentication process succeeds, the principal's X509v3 certificate status will be verified via the configured OCSP Responder.

Only when all credentials for the principal have been considered with no successful authentication will the service raise an AuthenticationFailure exception.

Authentication and runtime scope KSSL defines the scope of authentication of a given principal as starting upon successful return of the kssl::Context::authenticate() operation and continuing to be in effect until the current transaction (i.e. the transaction in which the authenticate() operation was called) commits, or the kssl::Context::endSession() operation is called for the given principal. All activity within the current transaction after authentication occurs is done on behalf of the authenticated principal.

Once authentication succeeds, access to protected resources is gated by the access control rules defined for the given principal in the corresponding protected resource object (see the section called “Access control services” on page 95). Access to resources (modeled types) for which no access control has been defined occurs with no security-related restrictions. All such activity, however, is logged for security audit purposes.

The access control service

The KSSL access control service provides the ability to restrict access to model elements. Access is defined by a set of access control rule directives, which may be specified via deployment specifications, or the KSSL API.

Access control is role-based. An access control rule grants or revokes access privileges to a given role, as applied to a given protected resource (i.e. model element).

Access control rules in deployment specifications

The syntax for defining an access control rule via a deployment specification is illustrated in the following example:

```
deploy myNode
{
    config Security
    {
        config AccessControl
        {
            config myPackage::myIfc
            {
                locked=true;
                lockAllElements=true;
                config Permissions
```

```
                {
                    admin=Create | Destroy;
                    endUser=Select;
                };
            };
        };
    };
};
```

In the above example, an access control rule for the `myPackage::myIfc` interface type specifies the following access directives:

locked=true	This directive specifies that all unauthenticated access to the interface be disallowed.
lockAllElements=true	This directive specifies that all operations and attributes of the interface be locked as well (i.e. blocks all unauthenticated access to those elements)
admin=Create Destroy	This directive specifies that the "admin" role has Create and Destroy privileges for the given interface.
endUser=Select	This directive specifies that the "endUser" role has Select privileges for the given interface.

Granting of privileges to roles occurs inside a Permissions block. A full description of all valid privileges follows later in this document.

Access control and type publication KSSL requires access to the shared memory type descriptor for a type which is to be protected. However, a typical user scenario includes loading of security policy before the components to be protected have been started (and consequently, before the referenced types have been published in shared memory).

To address this issue, KSSL provides a deferred security policy loading mechanism, which saves policy directives for types which are not found, and waits for the publication of those types. Once the types are published, the security policy directives are applied.

This deferred policy loading mechanism relies on publication of type descriptors for the referenced types. IDLos packages and modules do not possess type descriptors in shared memory, so deferred policy directives for packages and modules is not supported.

Access control rule scope KSSL supports specification of access control rules for the following model elements:

- interfaces
- attributes of interfaces
- operations in interfaces
- relationships defined for interfaces
- extents of interfaces (ability to “select” instances of the type)
- keyed lookup of interfaces (ability to “select ... where <key=value>”)
- module - applies to all interfaces in the module, recursively

- package - applies to all modules/interfaces in the package, recursively

No support is provided for access control on entities. Access to entities, which is already protected by package component boundaries, is considered secure from within a given component.

For the supported model elements defined above, KSSL allows users to specify access control rules for all instances of the given type (i.e. the extent) and/or for specific instances of the given type.

Access control and model elements The following table specifies the KSSL access control rules which may be applied to IDLos model elements. In the table, an `X' indicates that the given rule is applicable to the given IDLos type.

Access Control Type	Interface	Attribute	Operation	Relationship Role
Create	X			
Destroy	X			
Read		X		
Write		X		
Execute			X	
Relate				X
Select	X			X

The Select access control type, when applied to interface types, specifies that a given type may be searched for via the "select ..." statement. Similarly, Select, when applied to a relationship role, specifies whether the given relationship role my be navigated using a "select ..." statement. Finally, Select, when applied to a key name, specifies that they given key may be used in a "select ..." directive.

When specifying Create, Destroy or Select privileges, the application may give a package or module scoped name, as well as an interface's scoped name, as the type to set the permissions for. If a package or module name is given, then the permissions being specified are set for all interfaces contained in that package or module.

Dynamic ACL definition—the access control service The KSSL access control service allows rules to be specified for a modeled interface type: the access control rules are applied to all instances of that type. However, the service also allows access control to be specified at the individual object instance granularity level, thereby providing maximum operational control to user applications.

Defining access control rules—the ACL interface Applications use instances of the ACL interface to define access control rules for a given role. These instances act as input to the AccessControlManager operational methods. As such, they may be reused, created, destroyed or otherwise handled by the application independently of the resources being protected.

The ACL interface contains the following attribute:

aclRole	The name of the role to which this ACL pertains.

The ACL interface contains the following operations:

grant()	Assign the given permission to the given role.
revoke()	Revoke the given permission from the given role.

check()	Determines if the given permission is granted in this ACL.
copy()	Makes a copy of the given ACL instance.
lockAllElements()	Locks/unlocks all elements in an interface.

ACLs and models—the AccessControlManager interface The AccessControlManager interface allows applications to specify access control rules, as defined in one or more ACL instances, for specific model elements. Access control rules may be set for all instances of a given type, or for specific instances.

KSSL implements the AccessControlManager type as a singleton, which may be referenced by user applications via the appropriate "create ... singleton ..." directive. A single instance of this type exists on a given KIS node, and is used to manage access control rules for all types residing on that node.

Where there is an access control conflict between the permissions defined by a given ACL object and the type definition of the object itself (for example, adding kssl::Write to an attribute that is defined as "read-only" in IDLos) the attempt to define the access control rule will fail.

Access control and object instances KSSL specifies that when a conflict exists between access control rules defined for a given type and access control rules defined for a specific instance of that type, the access control rules specified for the object instance take precedence.

KSSL restricts access control for specific object instances to interface types which are the only interface exposing the implementing entity.

Access control and inheritance Access control rules which apply to a given interface, its attributes and/or operations also apply to derived interface types. For example, given the following sample interfaces:

```
interface Player
{
    attribute string name;
};
interface Forward : Player
{
    attribute long goals;
};
```

A role which has no Read permission for "Player::name" cannot access that attribute from instances of either "Player" or "Forward".

Similarly, a role which has no Create permission on type "Player" may not create an instance of "Forward" even if it has Create privileges for that derived type.

Access control and IDLos properties KSSL allows applications to define access rules that may be also defined via IDLos properties ("createaccess", "extentaccess", etc). KSSL specifies that access control rules specified via IDLos properties are in effect unless they are explicitly overridden via an KSSL-specified access control rule.

Access Control Runtime Scope KSSL guarantees enforcement of the access control rule specified via an AccessControlManager entry point (either setTypePerms() or setInstancePerms()) once that call has returned with no error indication.

Access control enforcement remains in effect until disableAccessControl() is called for the corresponding model element.

Access to protected resources The access control service enforces access control rules when protected resources are accessed by authenticated principals.

The access control service, however, also allows applications to define access control rules for unauthenticated access to a protected resource. Applications can specify such rules by passing the kssl::GlobalRole as the role value when defining access control rules. Using this facility, an application may revoke all privileges to unauthenticated users, thereby ensuring that only authenticated users access a given resource.

Access control exception conditions The access control service will carry out the following actions when an attempt is made to violate an access control rule:

- Write a descriptive message to the trace file for the engine where the access was attempted.
- Carry out any audit service notification requested by applications for the resources being accessed.
- Throw a DSEBuiltin::ExceptionAccessViolation system exception. If not caught by the action which carried out the access attempt, this exception will abort the current transaction.

The security audit service

The KSSL audit service provides users with the ability to trace all security-related actions taken by a principal, or involving a given protected resource. The service further provides the ability for the user application to receive callback notification of audit exception conditions at runtime, thereby allowing the application to initiate corrective action for the exception condition.

Audit events KSSL supports auditing of the following security events:

Authentication	Requests to authenticate principals.
Permissions	Requests to modify access control rules.
Lifecycle	Creation/destruction of protected resources.
Access	Access to attributes or invocation of operations which have access control rules defined.
All	all of the above.

Users of the audit service can request any combination of the above audit services. Users may also request that only error conditions or all events for the given audit type be logged.

Users may request audit services for all type elements for which access control rules can be specified, including specific instances of a given type.

Requesting audit services—the AuditManager interface The AuditManager interface provides entry points which allow users to specify audit policy for a given node. KSSL implements this type internally as a singleton. Applications may acquire an object reference to this object via the corresponding "create ... singleton" directive.

The AuditManager interface contains the following operations:

enableAuditPrincipal()	Enable audit of a given type of activity for the given principal.
disableAuditPrincipal()	Disable audit of the given type of activity for the given principal.

enableAudit()	Enable audit of the given type for the given user type or instance.
disableAudit()	Disable audit of the given type for the given user type or instance.
enableGlobalAudit()	Enable the given audit type for all activity in this node.
disableGlobalAudit()	Disable the given audit type for all activity in this node.
enableNotification()	Install an AuditEventNotifier instance to receive callback notification of the given audit event type.
disableNotification()	Disable audit event callback notifications.

Audit activity logging The KSSL Audit Service generates trace messages describing audit events as requested via the AuditManager interface. These messages are written to the engine's trace file if the Security trace bit is set for that engine.

Audit violation notification Applications may register a callback object to receive audit violation notifications from the KSSL Audit Service. The type of this callback object must inherit from the AuditEventNotifier abstract interface. The type must implement the notify() operation defined in that abstract interface. This operation will be invoked on the registered callback object whenever the audit condition specified when the notifier was installed occurs.

Users who request callback notification of all audit events should be aware that this may result in a high number of notification events to be delivered to their notifier object.

The callback notification operation includes parameters which describe the audit event which has occurred, the principal and model elements involved, and an informational message.

Audit event notification target objects must remain in the system while audit notification is enabled for a given node.

Secure access to KIS nodes

KIS utilizes KSSL services to secure access to node administration. Node security is based on the following role:

nodeAdmin Principals with nodeAdmin privileges can administer a node.

Administrative access to a node—execution of commands supported by the swsrv and swnode tools—is permitted only to users which have nodeAdmin role privileges.

The node creation process—the `swsrv -c start` command—carries out the following security-related node configuration:

- Define a principal for the calling OS (i.e. UNIX login) user, granting nodeAdmin privileges to that principal.
- Define the local machine as a Trusted Host.

The principal who creates the node (the OS user who executes the `swsrv -c start` command) becomes the owner of the node, by virtue of having nodeAdmin role privileges. That principal may delegate nodeAdmin privileges to additional principals by loading deployment specifications which define additional principals which have nodeAdmin role privileges.

The swsrv and swnode tools verify that the calling user is defined as a principal with nodeAdmin privileges before allowing the given command to continue.

Secure access to KSSL services

KSSL services provide entry points which may be used at runtime to alter the security policy in effect for a given node. To ensure that access to these services occurs only from trusted application code, KSSL grants access to KSSL API operational entry points only to principals who have nodeAdmin role privileges.

Secure registry access

The swregistry tool supports KSSL role-based access control to protected data, and confidentiality of protected data values, as follows:

- Support for locking of keys and/or values.
- Support nodeAdmin role-based access control, used to provide secure access to locked values.
- Support for encryption of locked values.

Locked data values are stored in an encrypted form. Access to these values is only allowed for OS (i.e. UNIX login) users who are defined as KSSL principals with nodeAdmin privileges on the node which the registry file configures.

There is no access control for registry instances that do not configure a node, since these instances have no security policy configuration.

The swregistry tool supports the following security-related command-line flags:

-l: Lock a key or value

-u: Unlock a key or value

Putting it all together: examples

The following code samples illustrates the use of the various KSSL service components by secure clients and services.

A simple secure service example

This example illustrates the use of KSSL to secure a service application. The sample defines the set of valid principals and associated roles for the service. It further specifies access control rules for a modeled resource, and requests some audit services.

Here is the IDLos for the service model:

```
//
// service.soc - A sample secure service application.
//
package myservice
{
    [local]
    entity admin
```

```
    {
        [initialize]
        void startup();
        void configureUsers();
        void configureResources();
        void login(
            in string userName,
            in string roleName,
            in string password);
    };

    interface Resource
    {
        attribute long x;
        string op();
    };
    entity ResourceImpl
    {
        attribute long x;
        string op();
    };
    expose entity ResourceImpl with interface Resource;
};

...

action myservice::ResourceImpl::op
{`
    printf("Resource::op(): Hello!\n");
    return "Hello";
`};
```

The following paragraphs describe each of the above operations in greater detail.

Authenticating a principal

In this example, the admin::login() action goes through the steps necessary to create a principal and its associated credential - using the simple text password mechanism - and authenticates that principal with the system.

```
action myservice::admin::login
{`
    declare
    {
        kssl::Status ret;
        kssl::Properties props;
        kssl::Credential cred;
        kssl::Role myrole;
        kssl::Principal user;
    }

    //
    // Create a Principal object for the
    // given user.
    //
    create user;
    user.name = userName;

    //
    // Initialize my credential to use the text
    // mechanism, and set the given password.
    //
    create cred;
    cred.type = kssl::MechText;
```

```
    cred.data = password;
    relate user hasCredential cred;

    //
    // Create a role.
    //
    create myrole;
    myrole.name = roleName;
    relate user hasRole myrole;

    //
    // Now authenticate the user.
    //
    try
    {
        user.authenticate();
    }
    catch (kssl::AuthenticationFailure ex)
    {
        declare string exMsg;
        exMsg = "Authentication failed : ";
        exMsg += ex.msg;
        printf(exMsg.getCString());
    }
`};
```

Configuring principals

The example's configureUsers() operation defines the principals that are valid users of the service, and their associated roles, as follows:

```
action service::driver::configureUsers
{`
    declare
    {
            kssl::Context           ctx;
            kssl::CredentialValue   cred;
            kssl::CredValues        pwds;
            kssl::RoleNames         roles;
            kssl::Properties        props;
    }

    //
    // Set up our user's credentials.
    //
    cred.name = "Rivaldo";
    cred.data = "numberXX";
    cred.type = kssl::MechText;
    pwds[0] = cred;
    cred.name = "Leonardo";
    cred.data = "numberYY";
    pwds[1] = cred;

    //
    // Set up and install the roles for our users.
    //
    roles[0] = "Midfielder";
    roles[1] = "Playmaker";
    roles[2] = "Maestro";

    //
    // now add the principal
    //
```

```
    ctx.addPrincipal("Rivaldo", roles, props, pwds);
`};
```

Configuring access control rules

The example's configureResources() operation creates the resources to be protected, specifies access control rules for those resources, and requests audit services, as follows:

```
action service::driver::configureResources
{`
    declare
    {
        Resource                        obj;
        X                               xobj;
        kssl::ACL                        acl;
        kssl::ACL                        opACL;
        kssl::ACL                        op2ACL;
        kssl::ACL                        attrACL;
        kssl::ACL                        attrACL2;
        kssl::AccessControlManager       mgr;
        kssl::Status                     ret;
    }

    create singleton mgr;

    //
    // Create an ACL, and populate it with type-level permissions.
    //
    create acl;
    acl.aclRole = "Playmaker";
    acl.grant(kssl::PermCreate);
    acl.grant(kssl::PermDestroy);
    acl.grant(kssl::PermSelect);

    //
    // add my ACL
    //
    mgr.addTypePerms("service::Resource", acl);

    //
    // Now define some access control for an attribute.
    //
    create attrACL;
    attrACL.aclRole = "Playmaker";
    attrACL.grant(kssl::PermRead);
    attrACL.grant(kssl::Write);
    mgr.addTypePerms("service::Resource::x", attrACL);

    //
    // Now define some revoked access control for an attribute.
    //
    create attrACL2;
    attrACL2.aclRole = "Playmaker";
    attrACL2.grant(kssl::PermRead);
    attrACL2.revoke(kssl::Write);
    mgr.addTypePerms("service::Resource::y", attrACL2);

    //
    // Now define some access control for an operation.
    //
    create opACL;
    opACL.aclRole = "Playmaker";
    opACL.grant(kssl::PermExecute);
    mgr.addTypePerms("service::Resource::op", opACL);
```

```
    //
    // Now define some revoked access control for an operation.
    //
    create op2ACL;
    op2ACL.aclRole = "Playmaker";
    op2ACL.revoke(kssl::PermExecute);
    mgr.addTypePerms("service::Resource::op2", op2ACL);

    //
    // Create a test obj and define some access control for it
    //
    create xobj;

    create acl;
    acl.aclRole = "Forward";
    acl.grant(kssl::PermDestroy);
    mgr.addInstancePerms(xobj, "service::X", acl);

    create acl;
    acl.aclRole = "Forward";
    acl.grant(kssl::PermExecute);
    mgr.addInstancePerms(xobj, "service::X::op", acl);

    create acl;
    acl.aclRole = "Forward";
    acl.grant(kssl::PermRead);
    acl.grant(kssl::Write);
    mgr.addInstancePerms(xobj, "service::X::l", acl);

`};
```

A simple secure client application

The following sample model illustrates a secure client which accesses the service illustrated in the previous sections. The client carries out authentication as one of the principals known to the service, and accesses the service's resources in a secure manner. Note that secure access to resources is transparent. That is, once authentication occurs, the actual business logic in the application contains no security-specific code.

```
package client
{
    [local]
    entity admin
    {
        [initialize]
        void    startup();

        //
        // Utility methods.
        //
        boolean     login(
                    in string userName,
                    in string password);
        void        fail(in string msg);
        void        msg(in string msgStr);
    };
};

action client::Test::login
{`
    declare
    {
        kssl::Credential     cred;
```

```
        kssl::Principal      user;
        boolean         myRet = true;
    }

    //
    // Create a Principal object for the
    // given user.
    //
    create user;
    user.name = userName;

    //
    // Initialize my credential to use the text
    // mechanism, and set the given password.
    //
    create cred;
    cred.type = kssl::MechText;
    cred.data = password;
    relate user hasCredential cred;

    //
    // Now authenticate the user.
    //
    try
    {
        msg("Calling user.authenticate() ...\n");
        user.authenticate();
        msg("user.authenticate() returned ok.\n");
    }
    catch (kssl::AuthenticationFailure ex)
    {
        declare string exMsg;
        exMsg = "Authentication failed : ";
        exMsg += ex.msg;
        exMsg += "\n";
        msg(exMsg);
        myRet = false;
    }

    return myRet;
`};

action client::admin::startup
{`
    declare
    {
        service::Resource           obj;
        boolean             ret;
    }
    msg("Starting testAuth() ...\n");

    //
    // Authenticate ourselves as a user of the service.
    //
    msg("Authenticating user...\n");
    ret = self.login("Rivaldo", "number10");

    if (ret)
    {
        msg("Authenticated user ok!\n");
    }
    else
    {
        self.fail("Failed to authenticate user!\n");
    }

    //
```

```
    // create the secure resource
    //
    create obj;

    //
    // access the object - access control will
    // be applied by kssl transparently
    //
    obj.x = 5;

    msg("All done.\n");
    SWBuiltIn::stop(0);
`};
```

Additional services and enhancements

The following sections describe additional KSSL services and enhancements used to develop, deploy, and manage secure KIS applications.

KSSL cryptography services

KIS applications may require a facility for ensuring data confidentiality for application-specific data values. The KSSL cryptography service provides such a facility via a component which presents interfaces used for data encryption, decryption and key agreement.

Cryptography services SDL The public interface to KSSL cryptography services is as follows:

```
    //
    // Cryptography services
    //
    enum Cipher
    {
        BlowFish,
        DES
    };

    enum CipherMode
    {
        CBCMode,
        ECBMode,
        CFBMode,
        OFBMode
    };

    enum MDType
    {
        MD5,
        SHA1
    };

    enum MACType
    {
        NONE,
        HMAC
    };

    exception OperationError
    {
        string  msg;
    };

    typedef string  KeyData;
```

```
interface CipherContext
{
    //
    // configuration methods
    //
    void    init(
            in Cipher       cipher,
            in CipherMode   mode);
    void    setKey(in KeyData keyData)
            raises (OperationError);
    void    setVector(in KeyData initVector);

    //
    // operational methods
    //
    void    encrypt(
            in string src,
            out string dest)
            raises (OperationError);
    void    decrypt(
            in string src,
            out string dest)
            raises (OperationError);
};

interface MDContext
{
    void    init(in MDType mdType);
    void    getMessageDigest(
            in string data,
            out string md);
    void    getMAC(
            in KeyData keyData,
            in string data,
            out string md);
};
```

Web/PHP adapter

The KIS Web/PHP adapter provides a set of API entry points which may be called from user PHP scripts. The entry points provide runtime access to user applications running on an KIS node.

KSSL specifies extensions to the Web adapter to support user authentication mechanisms provided by PHP environments. KSSL also specifies extensions to the Web Adapter to support confidential network communication between the PHP component (the os_*() functions) and the Web Adapter service itself.

PHP user authentication The Web Adapter utilizes two external services - An Apache server hosting the PHP module, and the PHP module itself. These services provide two basic user authentication mechanisms:

Web server security	This mechanism specifies that the web server (Apache) manage user authentication internally, by mapping user name/password pairs received from the browser to a prebuilt, encrypted database of user information. This mechanism provides no access to the authentication process from within a PHP script, and consequently does not provide the ability to carry out external authentication. Consequently this mechanism cannot be used in

	environments which wish to integrate PHP-based user applications with KIS security.
HTTP authentication	This mechanism provides PHP scripts with the ability to request user name/password information from the browser client, and to have access to those values within the script.

Both of the mechanisms described above provide support for simple text credentials. No other credential format is currently supported.

KSSL specifies that users who wish to integrate authentication of web clients with KIS security configure their Apache web servers to use HTTP authentication.

KSSL further requires that user PHP scripts include a header code block which will ensure that user name and credential information is requested from the browser client.

Finally, KSSL specifies extensions to the Web Adapter public interface which will convey the user name and credential information to the Web Adapter engine. The Web Adapter engine is required to carry out user authentication by the standard KSSL mechanisms, and to communicate the result to the calling PHP script.

Accessing user credentials from PHP scripts PHP scripts can require that the browser client provide user name and credentials by adding the following code block to the beginning (i.e. before calls to the Web Adapter interface) of their script:

```
<?php
 if(!isset($PHP_AUTH_USER))
 {
    Header("WWW-Authenticate: Basic realm=\"My Realm\"");
     Header("HTTP/1.0 401 Unauthorized");

    echo "Text to send if user hits Cancel button\n";
     exit;
 }
?>
```

After the above code segment executes, the `$PHP_AUTH_USER` and `$PHP_AUTH_PW` script variables will contain the user name and credentials, respectively.

Authenticating Web Adapter clients KSSL specifies security extensions to the os_connect() API entry point for the Web Adapter PHP component, which is now defined as follows:

```
conn_id os_connect([host], [port],
           [userName], [credential], [mechanism]);
```

This call allows users to convey the user name and password obtained from the browser client via the script segment illustrated above. In practice, users pass the PHP variables $PHP_AUTH_USER and $PHP_AUTH_PW as the "userName" and "password" parameter values, as in the following example:

```
...
/* Connect to KIS-land */
conn_id = os_connect("myHost", 555,
           $PHP_AUTH_USER, $PHP_AUTH_PW, "text");
```

The above call will convey the user name and password to the OSW service, where KSSL authentication will take place. If the user is authenticated, a valid connection identifier is returned. If authentication fails, and error is reported via the standard Web Adapter error handling mechanism.

6

Accessing KIS through PHP

PHP is a widely-used general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. The functionality of PHP can be extended by building dynamically loaded extensions.

The Kabira Infrastructure Server (KIS) PHP4 extension provides access to KIS objects from within PHP scripts. This lets you invoke requests on a web server that interact directly with KIS applications.

KIS also has a built-in PHP interpreter, so you can execute PHP scripts from within KIS actions.

This chapter begins with several sections that describe how to use the PHP extensions generally:

The chapter continues with a detailed description of each function in the KIS PHP extension:

The chapter finishes with the following sections that describe specific aspects of the PHP extension and provide an example:

Overview

The KIS PHP4 extension provides a set of PHP functions to access KIS applications using PHP scripts. The scripts can be executed in any of several ways:

- from a web browser
- from a command line using the CGI version of PHP
- from within a KIS action

The following table presents a brief description of each function call in the PHP extensions.

PHP extension function	Description
os_connect	create a new connection to the KIS Web Adapter engine
os_create	creates a KIS entity and returns its reference
os_delete	delete a KIS entity
os_disconnect	disconnect from the KIS engine
os_extent	retrieve the extent of object handles for a type
os_get_attr	retrieves the value of all attributes in a KIS entity
os_invoke	invoke an operation on a KIS entity
os_relate	relate two interfaces
os_role	retrieves an array of handles by navigating across a relationship
os_set_attr	set the value of a KIS attribute
os_unrelate	un-relate two interfaces

For more detailed information on each of the function calls see the reference in Chapter 12.

Data types

By nature, PHP is a loosely typed system and KIS is a strongly typed system. Therefore, there is no exact type mapping between PHP and KIS. The KIS PHP extension supports basic types, enums and object references. It also supports arrays and sequences of basic types, enums and object references. All variables default to type string. At runtime, PHP decides the variable's type by its context.

Basic types

The table below lists the basic types that are supported. The following sections provide details of how each of the types is supported.

KIS types	PHP types
short, long, unsigned short, unsigned long, long long, unsigned long long, float, double, char, octet, string, wchar, wstring	string

Boolean

In PHP the boolean value for false is 0 or "" (empty string) and all other values are true. The tables below show the mapping between KIS and PHP boolean values.

These KIS booleans...	map to these PHP booleans
SW_FALSE	0
SW_TRUE	1

These PHP booleans...	map to these KIS booleans
0 (zero) or "" (empty string)	SW_FALSE
All other values	SW_TRUE

Enum

The symbolic name of enums are used in PHP.

Object references

References to KIS objects in PHP are represented by strings. The internal format of the string is not documented. Do not manipulate object references directly. One special case is when the object reference represents an empty object. It is "" (empty string).

Arrays and sequences

KIS arrays, sequences and bounded sequences are mapped to PHP arrays. PHP arrays are truly dynamic. It does not have any size limitation nor do the elements have to have the same type.

When arrays are used with the KIS PHP extension the elements must be either:

- basic type

- enum
- object reference

You must keep in mind that KIS has bound checking for arrays and bounded sequences.

Unsupported types

The following KIS types are not supported.

- struct
- union
- any
- native

PHP script language

The KIS PHP4 extension must be executed from within a PHP script. This section briefly discusses how to write PHP scripts.

PHP syntax

The following is a PHP script example

```
<html><head><title>PHP Test</title></head> <body> <?php echo "Hello World<p>"; ?>
</body></html>
```

The code between <?php and ?> is PHP script. A PHP script need not be embedded inside of HTML file. It could simply be

```
<?php echo "Hello World<p>"; ?>
```

PHP script is free syntax and loosely typed language. All variables used in the script must have a prefix of dollar sign “$”. All statements must end with “;”.

```
<?php
    $name = “Jone Doe”;
    $num = 5;
    $flag = true;
    $list = array (1, 2, 3, 4);
    $attrList = array(“m_name” => “John”, “ssn” =>55555555);
?>
```

Using the extension

You must call the function os_connect() before using any other KIS extension functions. This gets a connection to the KIS Web Engine and subsequent requests are processed by Web Adapter.

```
<?php
    $conn_id = os_connect(“localhost”, 7654);
    $objr = os_create($conn_id, “myPkg::myIfc”);
```

```
    $msg = os_invoke($conn_id, $objr, “getMessage”);
    print $msg;
    os_delete($conn_id, $objr);
    os_disconnect($conn_id);
?>
```

Error handling

The error handling behavior in PHP is controlled by the settings in php.ini file.

```
error_reporting                          bit mask controls what to report
display_errors                         On or Off
track_errors                          On or Off
```

If display_errors is on, error message will show up on the web page. If track_errors is on, when error happens the message will also be set in a PHP variable called $php_errormsg. The following script segment shows how to use $php_errormsg

```
<?php
    $php_errormsg = ““;
    $conn_id = os_connect(“localhost”, 7654);
    if ($php_errormsg != ““)
    {
        print “Connect trouble: $php_errormsg\n”;
        exit(-1);
    }
    ...
?>
```

More information on standard PHP and HTML scripting languages can be found in any PHP or HTML manual. Or visit the PHP web site at http://www.php.net

PHP4 extension

This section provides a detailed description of each function in the KIS PHP extension. This information can also be found in Chapter 11.

os_connect

Creates a new connection to KIS Web Adapter engine.

Syntax

```
$conn_id =
os_connect ($host, $port, $principal, $credential, $mechanism, $mechanismdata);
$conn_id =
os_connect ($host, $port);
$conn_id =
os_connect ($host);
$conn_id =
os_connect ( );
```

Description

This function creates a connection between the PHP process and the Web Adapter engine.

conn_id is a PHP variable.

host and port are optional parameters that locate the Web Adapter engine.

If host and port are not set they will default to the host and port specified in the php.ini file.

principal, credential, mechanism, and the optional mechanismdata are used for secure connection to an engine, where mechanism specifies the type of authentication to be used for the principal and credential. If mechanism is speciifed as X509v3, mechanismdata must contain a path to a PEM encoded certificate file.

Warnings

None

Error conditions

- Not able to connect to KIS server

os_create

Creates a KIS object and returns its reference.

Syntax

```
$objr =
os_create ($conn_id, $scopedName, $attrList);
$objr =
os_create ($conn_id, $scopedName);
```

Description

This function creates an object and returns its reference in the variable objr. If the object has attributes, an optional attrList could be passed in to set the attribute initial values.

When the scopedName type is a singleton, this call acts like the create singleton in action language. It creates the singleton if it does not exist. Otherwise, it returns the object reference.

conn_id is the value returned by os_connect().

objr is the return value.

scopedName is a fully scoped KIS interface name.

attrList is an optional associative array of attribute names and values.

Example

```
//
// create a sales person with “name” initialized
//
$salesType = “myPackage::Salesman”;
```

```
$salesAttrArray = array('name' => 'Slick Willy');
$salesHandle = os_create($conn_id, $salesType, $salesAttrArray);
//
// create a customer - without attribute array
//
$custType = "myPackage::Customer";
$custHandle = os_create($conn_id, $custType);
```

Error conditions

- Not able to connect to KIS server
- Invalid scoped name
- Invalid attribute name
- Unsupported type
- No create access
- Duplicate Key

os_delete

Deletes a KIS object.

Syntax

```
os_delete($conn_id, $objrList);

os_delete($conn_id, $objr);
```

Description

The second parameter may be a either a single object reference or an array containing a list of object references.

conn_id is the value returned by os_connect().

objrList is an array of valid object instances to delete.

objr is a valid object instances to delete.

Warnings

None.

Example

```
//
// Delete all Orders
//
$sn = "mypackage::Order";
$orderList = os_extent($conn_id, $sn);
```

```
os_delete($conn_id,$orderList);
//
// delete a single object
//
$objr = os_create($conn_id, $sn);
os_delete($conn_id, $objr);
```

Error conditions

- Not able to connect to KIS server
- Invalid handle
- No delete access

os_disconnect

Disconnect from the KIS engine.

Syntax

```
os_disconnect ($conn_id);
```

Description

conn_id is the return value from a call to os_connect().

Warnings

None.

Example

```
/* close a connection to the Web Adapter engine */
os_disconnect($xconn);
```

os_extent

Retrieve the extent of object handles of a given type.

Syntax

```
$objrList =
os_extent($conn_id, $scopedName, $attrList);
$objrList =
os_extent($conn_id, $scopedName);
```

Description

This function will select objects of a given type. If attrList is provided, it contains a list of name-value pairs that are ANDed together as a where clause to filter the number of object handles being returned.

If the objects are keyed and attrList has key coverage, a keyed lookup will be performed.

conn_id is the value returned by os_connect().

objrList is the return value and is a PHP array of scalar values.

scopedName is a string containing the fully scoped name of a type.

attrList is an optional associative array of attribute names and values.

Warnings

None.

Example

```
//
// get all customers
//
$sn = “myPackage::Customer”;
$custList = os_extent($conn_id, $sn);
//
// return all customers in California
//
$sn = “myPackage::Customer”;
$whereClause = array(‘state’ => ‘CA’);
$custList = os_extent($conn_id, $sn, $whereClause);
```

Error conditions

- Not able to connect to KIS server
- Invalid scoped name
- Invalid attribute name
- Unsupported type or bad value
- No extent access

os_get_attr

Retrieves the values of the attributes in a KIS object.

Syntax

```
$attrList =
os_get_attr($conn_id, $objr, $filter);
```

```
$attrList =
os_get_attr($conn_id, $objr);
```

Description

This function retrieves attribute values from an object. If filter exists, only the values of those attributes named in the filter array will be returned. Otherwise, all attributes values will be returned.

conn_id is the value returned by os_connect().

attrList is an optional associative array of attribute names and values.

objr is a KIS object handle.

filter is an optional associative array of attribute names and values.

Warnings

None.

Example

```
//
// get all attributes of a cusotmer
//
$attrs = os_get_attr($conn_id, $custHandle);
for  ( reset($attrs);  $name = key($attrs); next($attrs ) )
{
        $avalue = $attrs[$name];
        print “$name = $value\n”;
}
//
// get the state attribute only
//
$filter = array ( “state” => “” );
$attrs = os_get_attr($conn_id, $custHandle, $filter);
$state = $attrs[“state”];
print “This customer is in state of $state\n”;
```

Error conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid attribute name

os_invoke

Invoke an operation on a KIS object.

Syntax

```
$returnValue =
```

```
os_invoke($conn_id, $objr, $opName, $param, $ex);
$returnValue =
os_invoke($conn_id, $objr, $opName, $param);
$returnValue =
os_invoke($conn_id, $objr, $opName);
```

Description

This function is used to invoke an operation on an object. The returnValue is not required on void operations. If a parameter is an inout or out parameter the value of that array element in param will be modified with the result parameter value after a call. If the operation raises user defined exception, $ex will contain the name of the exception type as a string upon return.

conn_id is the value returned by os_connect().

returnValue is the return value of the operation upon completion.

objr is a valid object instance handle.

opName is the name of an operation.

param is a nested associative array of parameter name and value pairs.

ex is the name of user exception thrown by the operation.

Example

```
//
// call a void operation that does not have any params
//
os_invoke($conn_id, $objr, “runtest”);
//
// call an operation with parameters that returns a boolean
//
$params = array (“name” => “Smith”, “number” => 5);
$ret = os_invoke($conn_id, $objr, “register”, $params);
if ($ret)
{
    print “OK\n”;
}
else
{
    print “register failed\n”;
}
```

When an operation raises user defined exceptions, os_invoke() can get the exception type, but not the exception data if it has member fields.

```
//
//  operation with user defined exceptions
//
$userex = ““;
$params = array();
os_invoke($conn_id, $objr, “myOp”, $params, $userex);
if ($userex != ““)
{
    print “Caught user exception $userex\n”;
}
```

Array and sequence types can be used as in, out, inout parameters and return values.

```
//
//  array or sequence type as out param
//
$params = array ( “myList” => array() );
$ret = os_invoke($conn_id, $objr, “getList”, $params);
$list = $params[“myList”];
for ($i = 0; $i < count($list); $i++)
{
    $value = $list[$i];
    print “list[ $i ] = $value\n”;
}
```

Error conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid operation name
- Invalid parameter name
- Unsupported type or bad value
- Application exception

os_relate

Relates two interfaces.

Syntax

```
os_relate($conn_id, $fromObjr, $roleName, $toObjr);
```

Description

This function relates two objects using the relationship role roleName.

conn_id is the value returned by os_connect().

fromObjr is a valid object reference handle.

roleName is the name of a role in a relationship between the “from” and “to” objects.

toObjr is a valid object reference handle.

Warnings

None.

Example

```
$salesType = “myPackage::Salesman”;
$salesAttrArray = array(‘name’ => ‘Slick Willy’);
$salesHandle = os_create($conn_id,$salesType, $salesAttrArray);
$custType = “myPackage::Customer”;
$custAttrArray = array( ‘name’ => ‘Lucent’, ‘state’ => ‘NJ’);
$custHandle = os_create($conn_id,$custType, $custAttrArray);
os_relate($conn_id,$salesHandle,  “hasCust”, $custHandle);
```

Error conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid role name

os_role

Retrieves an array of handles by navigating across a relationship.

Syntax

```
$objrList =
os_role($conn_id, $objr,  $roleNam, $attrList);
$objrList =
os_role($conn_id, $objr,  $roleNam);
```

Description

This function returns an array of object references as it navigates across the appropriate relationship. The name-value pairs in attrList are ANDed together to act as a where clause to filter the number of object handles being returned.

conn_id is the value returned by os_connect().

objrList is a list of object references.

objr is a KIS object handle.

roleName a fully scoped relationship role name.

attrList is an optional associative array of attribute names and values.

Warnings

None.

Example

```
/*
** assume Salesperson is related 1:M with customer
** and that $salesHandle is already populated with a
** valid Salesperson.
*/
$cList = os_role($conn_id, $salesHandle, “packageName::relationshipName::sellsTo”);
for ( reset($cList);  $cust = current($cList); next($ctList))
{
        /* do something with $cust */
}
```

Error conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid role name
- Invalid attribute name
- Unsupported type or bad value

os_set_attr

Sets a value in an attribute of a KIS object.

Syntax

```
os_set_attr($conn_id, $objr, $attrList);
os_set_attr($conn_id, $objr, $name, $value);
```

Description

This function sets a value of an object attribute. More than one attribute value can be set in an object.

conn_id is the value returned by os_connect().

objr is a KIS object handle.

attrList is an optional associative array of attribute names and values.

Example

```
//
// set the name of a customer
//
$attrArray = array(‘name’ => ‘John’);
os_set_attr($conn_id, $objr, $attrArray);
//
// or using
```

```
//
os_set_attr($conn_id, $objr, “name”, “John”);
//
// set array attribute
//
$value = array (1, 2, 3, 4, 5);
$attrArray = array (“longList” => $value);
os_set_attr($conn_id, $objr, $attrArray);
```

Error conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid attribute name
- Attribute is readonly
- Unsupported type or bad value
- Duplicate Key

os_unrelate

Unrelates two objects.

Syntax

```
os_unrelate($conn_id, $fromObjr, $roleName, $toObjr);
```

Description

This function unrelates two objects that are currently related using the relationship role roleName.

conn_id is the value returned by os_connect().

fromObjr is a valid object reference handle.

roleName is a relationship role name.

toObjr is a valid object reference handle.

Warnings

None.

Example

```
$custList = os_role($conn_id, $salesHandle,  “:sellsTo”);
for (reset($custList); $cust = current($custList); next($custList))
{
```

```
        os_unrelate($conn_id, $salesHandle,  'hasCust', $cust);
}
```

Error conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid role name

Web Server—Apache

When a web browser executes a PHP script that uses the KIS PHP extension, it needs a web server that is PHP enabled and has KIS extension loaded. Various web servers can be used with PHP, but Kabira has only certified the PHP extensions using the Apache server.

Figure 6.1 illustrates the flow of information between the web browser and the KIS application.

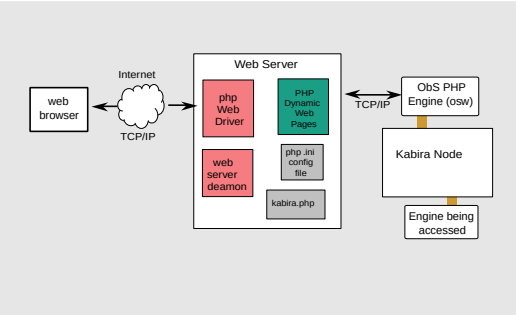

Figure 6.1. PHP extension architecture

Apache and PHP software are freely available. In order to build Apache and PHP, a number of GNU tools are required.

Please refer to $SW_HOME/OS.README.<version> file for details of how to build Apache and PHP.

Refer to $SW_HOME/OS.3RDPARTY.<version> for the supported version of Apache and PHP software.

Command Line Utility

A shell script wrapper for the CGI version of PHP is shipped with the release. The script name is "swphp".

Usage

```
Usage: swphp [-p path] [-g var=value] script_file
    -p     Set runtime path (default .)
    -g     Set init value for $var in script.
    Note: -g option must appear after all other options.
```

Like PHP scripts executed in a web browser, swphp also connects to the Web Adapter engine to process KIS extension functions. The swstart command creates a php.ini file in the current directory. It has the settings for swphp to start up the KIS extension within PHP.

The -p option tells swphp the directory path of where swstart was run. If it is executed in the same directory as swstart, -p option can be omitted.

The -g option sets variable initial values in the PHP script.

Example

```
swphp -g sn=myPkg::myIfc  -g opName=doIt run.php
<? php
    // File: run.php
    //
    // no need to give host and port, swphp will find the
    // right host and port from the registry.
    //
    $x_conn = os_connect();
    //
    // the value of $sn and $opName are set by -g option of swphp
    //
    $objr = os_create($x_conn, $sn);
    $msg = os_invoke($x_conn, $objr, $opName);
    os_delete($x_conn, $objr);
    print "calling $sn $opName returns $msg\n";
?>
```

Execute PHP in action language

This capability is also known as “PHP callout”. The PHP interpreter is modeled in the swmetadii package, an interface called PHP. The implementation is built into Web Adapter engine.

```
package swmetadii
{
    interface PHP
    {
        void executeScript(
            in    string scriptFile,
            in    NameValueList inParamList,
            inout NameValueList outParamList)
                raises (Failed);
        void evalString(
            in    string scriptString,
            in    NameValueList inParamList,
            inout NameValueList outParamList)
                raises (Failed);
    };
};
```

To build a KIS application that uses PHP callouts, you need to import the osw component in your build specification as shown below.

```
        component MYCOMP
        {
            import osw;
            ...
        };
```

The following action language segment shows how to use this component to perform callouts to PHP scripts.

```
declare swmetadii::PHP        caller;
create caller;
declare swmetadii::NameValueList    paramlist1;
declare swmetadii::NameValueList    paramlist2;
declare swmetadii::NameValue        param;
param.name = "VarName";
param.value = "VarValue";
paramlist1[0] = param;
param.name = "outmsg";
param.value = "";
paramlist2[0] = param;
declare string scriptFile  = "test.php";
try
{
    caller.executeScript(scriptFile, paramlist1, paramlist2);
    param = paramlist2[0];
    if (param.value != “passed”)
    {
        // something went wrong.
    }
}
catch (swmetadii::Failed e)
{
    msg = e.emsg;
    fprintf(stderr, "php call failed [%s]\n",msg.getCString());
}
```

The content of the PHP script “test.php” might look like the example below:

```
<?php
    if ($VarName == "VarValue")
    {
        print “in param has correct value\n”;
        $outmsg = “passed”;
    }
    else
    {
        print “in param does not have correct value\n”;
        $outmsg = “failed”;
    }
?>
```

Transactions

This section describes how transactions are managed when using the PHP extensions.

The transaction boundary varies, depending on whether PHP scripts are executed in a web browser, from the command line, or from action language.

Web browser or command line transactionality

In a web browser or from a command line, each PHP extension function call creates its own transaction. For instance,

```
$objr = os_create($conn_id, “myPkg::myIfc”);
```

When an os_create() call returns to the PHP script, two things have already happened:

- an object has been created
- the transaction of creating the object has committed

The next call that uses $objr to invoke an operation will be in a separate transaction.

The only way to ensure transactionality over a series of calls is to wrap the calls in a single function in the KIS model, then invoke that one function from PHP. Suppose you want to write a PHP script such as:

```
os_create(....);
os_set_attr(...);
os_relate(...);
...
```

If you want to make all of these calls execute in the same transaction, you need to model these functions as a single operation in your KIS application, for example:

```
action create_and_relate()
{‘
    create obj;
    obj.name = “John”;
    relate obj ... ;
    ...
‘};
```

These functions will now be performed as a single transaction by invoking the operation with os_invoke():

```
os_invoke(.., "create_n_relate()", ...);
```

Action language callout transactionality

When a PHP script is executed from within action language, it inherits the transaction environment from the caller object. The extension functions do not start their own transactions. Therefore the whole script is executed in the same transaction.

A PHP example

This example demonstrates how to create object references, set and get attributes, create and traverse relationship roles, and invoke operations on an object using a PHP script.

There is a model with customers and orders, and PHP scripts to create new customers, create new orders, and get customer balances.

The model

This section shows the UML and IDLos versions of the model used in this example

The following graphic shows the in UML model..

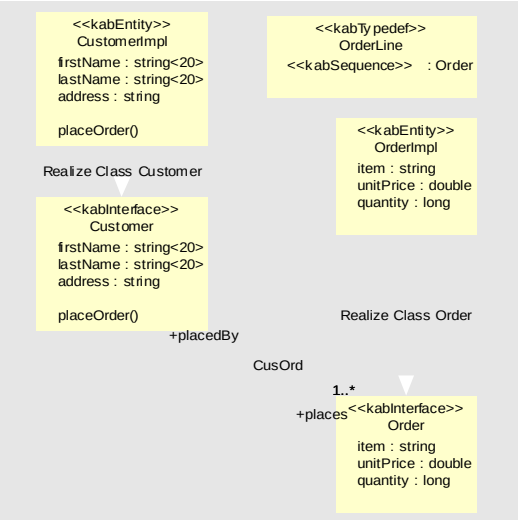

Figure 6.2. PHP example model

The following is the IDLos version of the model.

```
package orderMngt
{
    interface Order {
        attribute string item;
        attribute double unitPrice;
        attribute long quantity;
    };
    entity OrderImpl {
        attribute string item;
        attribute double unitPrice;
        attribute long quantity;
    };
    expose entity OrderImpl with interface Order;
    typedef sequence<Order>    OrderLine;
    interface Customer {
        attribute string<20> firstName;
        attribute string<20> lastName;
        attribute string address;
        key PKey { firstName, lastName };
        boolean placeOrder(inout Order ord);
    };
    entity CustomerImpl {
        attribute string<20> firstName;
        attribute string<20> lastName;
        attribute string address;
        key PKey { firstName, lastName };
        boolean placeOrder(inout Order ord);
        action placeOrder {`
            if (empty ord)  {
                return false;
            }
            relate self places ord;
            return true;
        `};
    };
    expose entity CustomerImpl with interface Customer;
    relationship CusOrd {
        role Customer places 1..* Order;
        role Order placedBy 0..1 Customer;
```

```
    };
};
```

Example scripts

This section shows the PHP scripts for the example.

add_customer.php This PHP script adds a new customer.

```
<?php
    $php_errormsg = “”;
    $conn_id = os_connect();
    if ( $php_errormsg != “”)
    {
        print “connect to OSW failed: $php_errormsg\n”;
        exit (-1);
    }
    $attrList = array (“firstName” => “John”,
                “lastName” => “Smith”);
    $typeName = “orderMngt::Customer”;
    $cus = os_create($conn_id, $typeName, $attrList);
    if ( $php_errormsg != “”)
    {
        print “add customer John Smith failed: $php_errormsg\n”;
        os_disconnect($conn_id);
        exit(-1);
    }
    print “Added new customer John Smith”;
    os_disconnect($conn_id);
?>
```

add_order.php This PHP script adds an order for an existing customer.

```
<?php
    $php_errormsg = “”;
    $conn_id = os_connect();
    if ( $php_errormsg != “”)
    {
        print “connect to OSW failed: $php_errormsg\n”;
        exit (-1);
    }
```

```
 $attrList = array (“firstName” => “John”,
    “lastName” => “Smith”);
 $typeName = “orderMngt::Customer”;
 $cusList = os_extent($conn_id, $typeName, $attrList);
 if ( $php_errormsg != “”)
 {
  print “find customer John Smith failed: $php_errormsg\n”;
        os_disconnect($conn_id);
        exit(-1);
    }
    if (count($cusList) == 0)    {
        print “did not find John Smith”;
        os_disconnect($conn_id);
        exit(-1);
    }
    $attrList = array (“item” => “Apple”,
                “unitPrice” => 0.99,
                “quantity” => 50);
    $typeName = “orderMngt::Order”;
    $ord = os_create($conn_id, $typeName, $attrList);
```

```
    if ( $php_errormsg != "")
    {
        print "create order failed: $php_errormsg\n";
        os_disconnect($conn_id);
        exit(-1);
    }
    $params = array ("ord" => $ord);
    $cus = $cusList[0];
    $status = os_invoke($conn_id, $cus, "placeOrder", $params);
    if (! $status)
    {
        print "place order for John Smith failed\n";
        os_disconnect($conn_id);
        exit(-1);
    }
    print "added order for John Smith\n";
    os_disconnect($conn_id);
?>
```

get_balance.php This script gets the current balance for an existing customer's account.

```
<?php
    $php_errormsg = "";
    $conn_id = os_connect();
    if ( $php_errormsg != "")
    {
        print "connect to Web Adapter failed: $php_errormsg\n";
        exit (-1);
    }
    $attrList = array ("firstName" => "John",
                "lastName" => "Smith");
    $typeName = "orderMngt::Customer";
    $cusList = os_extent($conn_id, $typeName, $attrList);
    if ( $php_errormsg != "")
    {
        print "find customer John Smith failed: $php_errormsg\n";
        os_disconnect($conn_id);
        exit(-1);
    }
    if (count($cusList) == 0)    {
        print "did not find John Smith";
        os_disconnect($conn_id);
        exit(-1);
    }
    $orderList = os_role($conn_id, $cusList[0], "places");
    $total = 0.0;
    for ($i = 0; $i < count($orderList); $i++)
    {
        $attrList = os_get_addr($conn_id, $orderList[$i]);
        $unitPrice = $attrList["unitPrice"];
        $quantity = $attrList["quantity"];
        $total += $unitPrice * $quantity;
    }
    print "send a bill to John Smith for $tota\n";
    os_disconnect($conn_id);
?>
```

Part Two

Part Two: KIS Reference

7

Lexical and syntactic fundamentals

This chapter describes the basic lexical rules of the Kabira Infrastructure Server (KIS) modeling language, IDLos. These rules follow the IDL (version 2.2) standard. These rules also apply to action language and build specifications.

The chapter covers the character set, tokens, white space, comments, pre-processing directives, and the syntax for properties.

Character set

IDLos supports the ISO Latin-1 (8859.1) character set. This consists of:

Decimal digits

These are: 0 1 2 3 4 5 6 7 8 9

Graphic characters

The graphic characters are: ! " # $ % & ' ( ) * + = . / : ; < = > ? @ [\ ] ^ _ ' { | } ~ , plus the extended set.

Formatting characters

The formatting characters are: BEL, BS, HT, LF NL, VT, FF, CR.

Alphabetic characters

The alphabetic characters are a-z and A-Z. plus these: àÀ áÁ âÂ ãÃ äÄ åÅ æÆ çÇ èÈ éÉ êÊ ëË ìÌ íÍ îÎ ïÏ ñÑ òÒ óÓ ôÔ õÕ öÖ øØ ùÙ úÚ ûÛ üÜ ÿ and the Icelandic thorn and eth characters, both capital and lowercase.

Tokens

There are five kinds of tokens: keywords, identifiers, literals, operators and other separators.

Keywords

The following table lists all the keywords reserved by IDLos.

Although this is a large table, there are only about 20 statements that need to be defined in the IDLos reference pages. IDLos really is a small language; many of the statements use two or three keywords, and there are 14 basic types of keywords. The trigger statement alone uses 11 keywords.

abort	fixed	relationship
abstract	float	role
action	from	select
annotation	ignore	sequence
any	in	singleton
attribute	initialize	short
boolean	inout	signal
by	interface	stateset
cannothappen	key	string
case	local	struct
char	long	switch
commit	module	terminate
const	native	to
context	Object	transition
create	octet	trigger
createaccess	order	true
default	oneway	TRUE
delete	out	typedef
deleteaccess	package	unsigned
double	post-get	union
during	post-set	unrelate
enum	pre-get	upon
entity	pre-set	using
exception	raises	void
expose	readonly	wchar
extentaccess	refresh	with
false	recovery	wstring
FALSE	relate	virtual
finished		

The keywords context, fixed are parsed, but are not used by KIS Design Center at this time.

Identifiers

An identifier is an arbitrarily long sequence of alphabetic, digit, and underscore ("_") characters. The first character must be an alphabetic character. Only one of the letters A-Z or a-z may be used as the first character. Identifiers are case sensitive.

Unlike IDL, identifiers in IDLos are case sensitive.

If you need to use a keyword as an IDLos identifier, you can delimit it with quotation marks, for example:

```
entity purchase
{
    attribute long amount;
    attribute boolean "commit";
}
...
declare purchase p;
create p;
p."commit"=FALSE;
```

You can use any IDLos or action language keyword as a quoted IDLos identifier, unless it is also a C++ or IDL reserved word. Be aware that in some cases this can lead to ambiguity, for example:

```
void myAction(in long "commit"); // quoted keyword as parameter
...
                     // if the action contains this expression
if ("commit" == 1)   // then "commit" is parsed as a string literal
                     // and not as the input parameter
```

Literals

Literals may be integer, character, floating point, string, or boolean:

integer	A sequence of numerals beginning with 0 (e.g. 077) is interpreted as an octal number. A sequence of numerals beginning with any other number (e.g. 63) is interpreted as a decimal number. A sequence of numerals prefixed with 0x or 0X (e.g. 0x3F) is interpreted as a hexadecimal number. Hexadecimals may use a-f or A-F. A sequence of numerals appended with LL is interpreted as a 64-bit integer (long long).
character	A character literal is enclosed in single quotes (e.g. 'Z'). Standard IDL escapes are allowed, such as '\n' for newline and '\x67' for hexadecimal rendition. There is no support for wide character literals at this time.
floating point	A floating point literal consists of an integer part, a decimal point, a fraction part, and e or E, and an optionally signed integer exponent (e.g. 1.024+e13). The decimal point or the

	exponent may be missing, but not both. The integer part or the fraction part may be missing, but not both.
string	String values are placed in double quotes (e.g. "This is a string"). Adjoining strings are concatenated to form long strings. Newlines and other characters must be embedded with escape sequences.
boolean	A boolean literal: TRUE, true, FALSE, or false.

Escape sequences

The following table defines the escape sequences

Description	Escape sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash	\\
question mark	\?
single quote	\'
double quote	\"
value of character in octal	\ooo
value of character in hexadecimal	\xhh

Operators

The operator tokens in IDLos are: - + ~ = ! && / * ( ) ?.

Other separators

Other separators in IDLos are:

- preprocessor token #
- action language delimiter ‘
- statement delimiter ;
- list delimiter ,
- inheritance delimiter :
- scoping delimiter ::

- block grouping { }
- list grouping ( )
- array definition, property grouping [ ]
- sequence declaration < >
- comment tokens // /* */
- role multiplicity delimiter ..

White space

White space is comprised of blanks, horizontal and vertical tabs, newlines, form feeds and comments. White space is ignored except as it serves to separate tokens. For example, when an inherited interface must be scoped

```
interface A : ::SomePackage::B {};    // correct
interface B :::SomePackage::B {};     // wrong! ::: causes error
```

Comments

Single line comments start with // and continue to a newline. C-style comments start with a /* and continue until a */, and may include newlines.

Comments do not nest. Once a comment has started, all comment delimiters are ignored except the end delimiter for that style of comment.

Preprocessing directives

The following preprocessing directives are supported. Preprocessing directives are not terminated by a semi-colon. Any other line where the first non-white space character is '#' is ignored.

#include <filename>	When #include is the first non-white space on a line, the text after the #include is interpreted as a file name and will be included at that point in the text. The file is processed as IDLos.
#pragma include <filename>	Include the specified file in the generated output. The file is not be parsed by the IDLos loader.
#pragma	All other #pragma statements are not parsed. They are loaded into the repository and exported as is by the exporter. This provides round-trip support for #pragma statements.

KIS IDLos supports #include "headerfile.h". The IDLos loader adds the local directory "." to the include path so there is no difference between #include <headerfile.h> and #include "headerfile.h". This differs from normal cpp rules for #include.

Quoted Keywords

IDLos allows keywords to be escaped by putting the identifier in double quotes. This allows reserved IDLos keywords to be used as normal identifiers if required.

For example, the following is legal IDLos:

```
//
//     Escape the reserved words interface and entity
//
interface "interface" { };
entity "entity" { };
expose entity "entity" with interface "interface";
```

8

IDLos types

This chapter describes the Kabira Infrastructure Server (KIS) type system.

Types are labels that specify what category of information a symbol indicates. Usually one category is not comparable with another (for example, a string is not comparable with a float). KIS uses the IDL (version 2.2) type system, which provides a rich system of basic and user-defined types.

For each type, there is a brief description of the semantics of the type, and, where appropriate, how to specify the type in IDLos and/or Action Language. That will be followed by one or more examples, and any general information.

The following types are supported by KIS. Simple types are marked with an asterix(*).

- the section called “any” on page 150*the section called “array” on page 151, the section called “boolean” on page 152*, the section called “bounded sequence” on page 152
- the section called “bounded string” on page 154, the section called “bounded wstring” on page 154, the section called “char” on page 155*
- the section called “const” on page 155, the section called “double” on page 156*, the section called “entity” on page 157, the section called “enum” on page 157
- the section called “exception” on page 158,
- the section called “long” on page 161*, the section called “long long” on page 161*, the section called “ native” on page 162
- the section called “Object” on page 162*, the section called “octet” on page 163*, the section called “sequence” on page 163, the section called “short” on page 165*
- the section called “string” on page 165*, the section called “struct” on page 166, the section called “typedef” on page 167, the section called “union” on page 167, the section called “unsigned long” on page 169*

- the section called “unsigned long long” on page 170*, the section called “unsigned short” on page 170*
- the section called “void” on page 171, the section called “wchar” on page 171*, the section called “wstring” on page 172

any

The type any is an IDL simple type. It can hold any IDLos type. An any logically contains a TypeCode, and a value that is described by the TypeCode.

IDLos syntax

Inside of an entity block, define an attribute “name” of type any with:

```
attribute any name;
```

Define an array “name” of type any with:

```
typedef any name[5];
```

Define a sequence “name” of type any with:

```
typedef sequence<any> name;
```

Define a bounded sequence “name” of type any with:

```
typedef sequence<any, 5> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type any with:

```
declare any name;
```

Inside of an action language function, allocate an array “name” of type any with:

```
declare any name[5];
```

Example

```
declare any myAny;
declare any myAny2;
declare string str;
declare string str2;
str = "A string lives here.";
myAny <<= str;
myAny2 = myAny;
myAny2 >>= str2;
```

General Information

Operator Descriptions:

equals;	Assigns one any to another any
<<=	Inserts a value into an any.

>>=	Extracts a value from an any.

The any is always the first operand of the insertion and extraction operators.

If there is a type mismatch between the value contained in an any and the target of an extraction operator, the following exception occurs:

```
swbuiltin::ExceptionAnyTypeMismatch
```

You should wrap an extraction operation in a try-catch block that catches the above exception.

```
action myPackage::myEntity::myOp
{`
     declare any myAny;
     declare string str;
     str = "A string lives here.";
     myAny <<= str;
     ...
     try
     {
         myAny >>= str;
         printf("%s\n", str.getCString());
     }
     catch (swbuiltin::ExceptionAnyTypeMismatch)
     {
         printf("A string doesn't live here any more.\n");
     }
`};
```

array

The type array is an IDL composite type. It represents a known-length series of a single type. An array must be named by a typedef declaration in order to be used as a parameter, or a return value. You can omit a typedef declaration only for an array that is declared within a structure or attribute definition.

IDLos syntax

See IDLos syntax for a simple type, such as any.

Action language syntax

See Action language syntax for a simple type, such as any.

Example

This struct defines 256x256 array of 8-bit octets to hold an image:

```
struct SmallImage
{
     octet pixel[256][256];
};
```

General Information

IDLos and Action Language support any number of dimensions. All dimensions in an array must be bounded. To define an array with unbounded dimensions, use a sequence.

boolean

The type boolean is an IDL simple type. It represents a data item that can only be assigned the values of true or false.

IDLos syntax

Inside of an entity block, define an attribute “name” of type boolean with:

```
attribute boolean name;
```

Define an array “name” of type boolean with:

```
typedef boolean name[5];
```

Define a sequence “name” of type boolean with:

```
typedef sequence<boolean> name;
```

Define a bounded sequence “name” of type boolean with:

```
typedef sequence<boolean, 5> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type boolean with:

```
declare boolean name;
```

Inside of an action language function, allocate an array “name” of type boolean with:

```
declare boolean name[5];
```

Example

```
declare boolean result;
result = true;
```

bounded sequence

The type bounded sequence is an IDL template type. Bounded sequences can hold any number of elements, up to the limit specified by the bound. A bounded sequence must be named by a typedef declaration in order to be used as a parameter, an attribute, or a return value. You can omit a typedef declaration only for a sequence that is declared within a structure definition.

Bounded sequences have a special `length` attribute that you use in action language to determine the current number of elements in the sequence (see the action language example below).

IDLos syntax

```
typedef sequence<any,5> name;
```

Action language syntax

Sequences cannot be defined other than as a typedef or as a member of a struct or exception.

Example

```
const long constBoundsValue = 50;
typedef string strArray[constBoundsValue];
typedef sequence<strArray,constBoundsValue> boundedSeqOfStrArrys;
struct Account
{
    string customer;
    string cardCompany;
    sequence <unsigned short, 10> creditCard;
};
struct LimitedAccounts {
    string<10> bankSortCode;
    sequence<Account, 50> accounts; // max sequence length is 50
};
```

Action language example

```
package test
{
   const long constBoundsValue = 10;
   typedef sequence<string, constBoundsValue> IndexList;

   [local]
   entity testEntity
   {
      [initialize]
      void doIt();
      action doIt
      {`
         declare long i;
         declare IndexList myList;
         for (i=1;i<=constBoundsValue;i++)
         {
            myList[myList.length] = i;
         }
         myList[5] = 5;
         printf("List values: ");
         for (i=0;i<constBoundsValue;i++)
         {
            printf(myList[i]);
            printf(" ");
         }
      `};
   };
};
```

The component outputs the following line to `coordinator.stdout` :

```
List values: 1 2 3 4 5 5 7 8 9 10
```

bounded string

The type bounded string is an IDL template type. It is a series of any number of possible 8-bit quantities, up to the limit specified by the bound.

IDLos syntax

You may define a bounded string anywhere you can define a simple type.

```
attribute string<1024> buffer;
```

Action language syntax

```
declare string<10> myString;
```

Example

```
struct ShortMessagePacket
 {
     string destination;
     string<1024> shortMessage;
 };
```

bounded wstring

The type bounded wstring is an IDL template type.

IDLos syntax

You may define a bounded wstring anywhere you can use a simple type.

```
 struct ShortMessagePacket
 {
     wstring destination;
     wstring<1024> shortMessage;
 };
```

Action language syntax

```
declare wstring<10> wideLabel;
```

For string assignments and implicit string conversion rules, see “Manipulating data” on page 104.

Example

```
struct ShortMessagePacket;
{
    wstring destination;
    wstring<1024> shortMessage;
};
declare wstring<1024> wideBuffer;
```

char

The type char is an IDL simple type. It is an 8-bit quantity that encodes a single-byte character from any byte-oriented code set. The type char can hold any value from the ISO Latin-1 character set. Code positions 0-127 are identical to ASCII. Code positions 128-255 are reserved for special characters in various European languages, such as accented vowels.

IDLos syntax

Inside of an entity block, define an attribute “name” of type char with:

```
attribute char name;
```

Define an array “name” of type char with:

```
typedef char name[5];
```

Define a sequence “name” of type char with:

```
typedef sequence<char> name;
```

Define a bounded sequence “name” of type char with:

```
typedef sequence<char, 5> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type char with:

```
declare char name;
```

Inside of an action language function, allocate an array “name” of type char with:

```
declare char name[5];
```

Example

```
declare char name;
```

const

Used in conjunction with other IDL types, const assigns a symbol to an unchanging value.

These types can be made const: boolean, char, double, float, long, long long, short, string, unsigned long, unsigned long long, unsigned short, wchar, wstring.

IDLos syntax

```
entity name
{
    const short parm = 5;
};
```

Example

```
const long numOfEnglishLetters = 26;
typedef string EnglishIndexURLs[numOfEnglishLetters];
entity htmlIndex
{
      attribute EnglishIndexURLs URLs;
     boolean getIndexURL(in long i, out string url);
     action getIndexURL
     {`
         if (i >= 0 && i < numOfEnglishLetters)
         {
             url = self.URLs[i];
             return true;
         }
         else
         {
             url = "";
             return false;
         }
     `};
};
```

General Information

There is no such thing as an any constant value.

A constant value can be specified as a literal value or a scoped name to another user defined constant value.

double

The type double is an IDL simple type. It represents IEEE double-precision floating point numbers. Types float and double follow IEEE specifications for single- and double-precision floating point values, and on most platforms map to native IEEE floating point types.

IDLos syntax

Inside of an entity block, define an attribute “name” of type double with:

```
attribute double name;
```

Define an array “name” of type double with:

```
typedef double name[5];
```

Define a sequence “name” of type double with:

```
typedef sequence<double> name;
```

Define a bounded sequence “name” of type double with:

```
typedef sequence<double, 5> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type double with:

```
declare double name;
```

Inside of an action language function, allocate an array “name” of type double with:

```
declare double name[5];
```

Example

```
declare double name;
```

entity

The type entity is an IDL composite type.

With an entity, you can:

- define attributes and operations
- inherit from other entities
- define any number of relationships with other entities
- make it accessible to other packages using an interface
- trigger operations when it is created, deleted, refreshed, updated, or related to another entity
- define keys to use in queries
- manage its states with a finite state machine
- define types nested in the entity

IDLos syntax

```
 entity name ( : <parent type > )
 {
        …
}
```

Example

```
entity TimerEventImpl
{
     // attributes
     attribute Road road;
     // operations
     oneway void generate ();
};
```

enum

The type enum is an IDL composite type.

An enum (enumerated) type lets you assign identifiers to the members of a set of values.

IDLos syntax

```
enum name { <list of values ( = const_expr )>;
```

Example

```
enum SideDirection {North, East = 5, South, West};
```

This defines SideDirection as a type which takes on the values North, East, South or West.

Action Language example:

```
declare enum SideDirection localDirection;
localDirection = East;
```

General Information

The value of an enum is not defined by IDLos other than IDLos guarantees that the ordinal values of enumerated types monotonically increase from left to right. The value of an enum can also be explicitly defined by the modeler using an initializer value. All enumerators are mapped to a 32-bit type.

exception

The type exception is an IDL composite type.

The IDLos exception type itself is similar to a struct type; it can contain various data members (but no methods). An exception can inherit from another exception. All attributes in the parent type are available in the child type.

An exception is used to alter the flow of control of a section of code. When a method raises an exception, the method "returns" at that point in its code. When the calling code receives the exception, control passes to any matching catch blocks, or the exception is rethrown.

Uncaught user and system exceptions will cause a runtime engine failure.

IDLos syntax

```
exception name ( : < parent type > ) { <attribute list> };
```

Example

```
exception IsDead { string causeOfDeath; };
void wakeup() raises (IsDead);
```

This defines the user exception IsDead as a type with attribute causeOf Death. The function wakeup() can instantiate a variable of type IsDead, set the value of causeOfDeath and throw the variable as a user exception. To catch the above exception:

```
try { self.wakeup(); } catch (IsDead id) {...}
```

General Information

User exceptions are defined in IDLos.

System exceptions are defined in swbuiltin.sdl. They can be raised by the KIS runtime and adapters. The user should not throw System exceptions, but should catch some of them in certain conditions. To catch system exceptions in action language, include swbuiltin.sdl in the model.

Handling the system exceptions ExceptionDeadlock and ExceptionObjectDestroyed in application code not executed as part of spawn will produce unpredictable but bad results.

Operations that throw an exception must have a raises clause in the operation signature.

An exception definition cannot include itself as a member.

float

The type float is an IDL simple type.

It represents IEEE single-precision floating point numbers. Types float and double follow IEEE specifications for single- and double-precision floating point values, and on most platforms map to native IEEE floating point types.

IDLos syntax

Inside of an entity block, define an attribute “name” of type float with:

```
attribute float name;
```

Define an array “name” of type float with:

```
typedef float name[<size>];
```

Inside of an entity block, define a sequence “name” of type float with:

```
typedef sequence<float> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type float with:

```
declare float name;
```

Inside of an action language function, allocate an array “name” of type float with:

```
declare float name[<size>];
```

Example

```
declare float name;
```

interface

An IDLos interface exposes an entity's operations and interfaces. An interface is the mechanism to make implementation visible outside of package boundaries.

An IDL attribute is short-hand for a pair of operations that get and, optionally, set values in an object.

IDLos syntax

```
interface Employee
{
     // only these parts of EmployeeImpl will be
     // accessible outside the package.
     void promote();
     oneway void makewaves();
     attribute string name;
};
entity EmployeeImpl
{
     void promote();
     attribute string name;
     attribute string m_nickname;
     signal makewaves();
};
```

Example

In this example, the EmployeeImpl entity has two attributes, an operation, and a signal. The interface exposes all but the m_nickname attribute. Notice that the signal is exposed as an operation.

```
interface Employee
{
     // only these parts of EmployeeImpl will be
     // accessible outside the package.
     void promote();
     oneway void makewaves();
     attribute string name;
};
entity EmployeeImpl
{
     void promote();
     attribute string name;
     attribute string m_nickname;
     signal makewaves();
};
// This statement is self-explanatory.
expose entity EmployeeImpl with interface Employee;
```

General Information

An interface may also contain definitions for typedefs, attributes, operations, enumerations, and structures.

Each interface may expose only one entity. An entity may be exposed by more than one interface. Each operation or attribute in the interface must have a corresponding attribute, operation or signal in the entity.

Abstract interfaces may not contain attributes or typedefs.

To permit more control over how an entity is exposed, three properties grant or revoke access to create, delete, or retrieve the extent of an interface.

long

The type long is an IDL simple type. It represents 32-bit signed integer values. IDL supports short and long integer types, both signed and unsigned. IDL guarantees the range of these types.

IDLos syntax

Inside of an entity block, define an attribute “name” of type long with:

```
attribute long name;
```

Define an array “name” of type long with:

```
typedef long name[size];
```

Define a sequence “name” of type long with:

```
typedef sequence<long> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type long with:

```
declare long name;
```

Inside of an action language function, allocate an array “name” of type long with:

```
declare long name[<size>];
```

Example

```
declare long name;
```

long long

The type long long is an IDL simple type.

The long long type represents 64-bit signed integer values.

IDLos syntax

Inside of an entity block, define an attribute “name” of type long long with:

```
attribute long long name;
```

Define an array “name” of type long long with:

```
typedef long long name[<size>];
```

Define a sequence “name” of type long long with:

```
typedef sequence<long long> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type long long with:

```
declare long long name;
```

Inside of an action language function, allocate an array “name” of type long long with:

```
declare long long name[<size>];
```

Example

```
declare long long name;
```

native

A native type is used to represent a platform specific type that is opaque to the IDLos type system.

IDLos syntax

Inside of an entity block, define an attribute “name” of type native with:

```
attribute native name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type native with:

```
declare native name;
```

Object

The type Object is an IDL simple type. It represents a reference to a user-defined interface or entity.

IDLos syntax

Inside of an entity block, define an attribute “name” of type Object with:

```
attribute Object name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type Object with:

```
declare Object name;
```

Example

```
declare Object name;
```

octet

The type octet is an IDL simple type.

It represents an 8-bit quantity guaranteed not to undergo any conversion when transmitted by the communication system. This lets you safely transmit binary data between different address spaces. Avoid using type char for binary data, inasmuch as characters might be subject to translation during transmission. For example, if client that uses ASCII sends a string to a server that uses EBCDIC, the sender and receiver are liable to have different binary values for the string's characters.

IDLos syntax

Inside of an entity block, define an attribute “name” of type octet with:

```
attribute octet name;
```

Inside of an entity block, define a sequence “name” of type octet with:

```
typedef sequence<octet> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type octet with:

```
declare octet name;
```

Inside of an action language function, allocate an array “name” of type octet with:

```
declare octet name[<size>];
```

Example

```
declare octet name;
```

sequence

The type sequence is an IDL template type. It represents an array with unbounded dimensions. Unbounded sequences are automatically grown at runtime as required. If the index being accessed is larger than the current sequence size, the sequence is resized. When a sequence is resized, the values of all new sequence members are undefined. They must be initialized to a known value by the application.

The index for sequences starts at `0` .

A sequence must be named by a typedef declaration in order to be used as a parameter, an attribute, or a return value. You can omit a typedef declaration only for a sequence that is declared within a structure definition.

Sequences have a special `length` attribute that you use in action language to determine the current number of elements in the sequence (see the action language example below).

IDLos syntax

```
typedef sequence<string> NameList;
```

Example

```
const long constBoundsValue = 50;
typedef string stringArray[constBoundsValue];
typedef sequence<stringArray> SeqOfStringArrays;
struct Account
{
    string customer;
    string cardCompany;
    sequence<unsigned short> creditCard;
};
struct UnlimitedAccounts
{
    string<10> bankSortCode;
    sequence<Account> accounts; // no max sequence length
};
```

Action language example

Assume you start an engine implemented as follows:

```
package test1
{
   typedef sequence<string> IndexList;
   const long constBoundsValue = 10;

   [local]
   entity testEntity
   {
      [initialize]
      void doIt();
      action doIt
      {`
         declare long i;
         declare IndexList myList;
         for (i=1;i<=constBoundsValue;i++)
         {
            myList[myList.length] = i;
         }
         myList[5] = 5;
         printf("List values: ");
         for (i=0;i<constBoundsValue;i++)
         {
            printf(myList[i]);
            printf(" ");
         }
      `};
   };
};
```

The component outputs the following line to `coordinator.stdout` :

```
List values: 1 2 3 4 5 5 7 8 9 10
```

General Information

Sequences cannot be defined other than as a typedef or as a member of a struct or exception. An attribute length represents the current length of the sequence at runtime.

short

The type short is an IDL simple type. It represents 16-bit signed integer values.

IDLos syntax

Inside of an entity block, define an attribute “name” of type short with:

```
attribute short name;
```

Define an array “name” of type short with:

```
typedef short name[<size>];
```

Define a sequence “name” of type short with:

```
typedef sequence<short> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type short with:

```
declare short name;
```

Inside of an action language function, allocate an array “name” of type short with:

```
declare short name[<size>];
```

Example

```
declare short name;
```

string

The type string is an IDL template type. It is a series of all possible 8-bit quantities. Unbounded string lengths are generally constrained only by memory limitations.

You can also provide allocation hints to the model compiler when declaring unbounded strings, by using the sizehint property on a typedef string statement. The modeler compiler allocates the requested size on the stack, which can improve performance over allocating memory in the heap to grow an unbounded string.

IDLos syntax

```
attribute string name;
```

Action language syntax

```
declare string name;
```

For string assignments and implicit string conversion rules, see “Manipulating data” on page 104.

Example

```
typedef string myString;
[ sizehint=250000 ]
typedef string MyPresizedString;

attribute myString shortName;
attribute myPresizedString anotherName;
```

General Information

An attribute length represents the current length of the string at runtime.

struct

The type struct is an IDL composite type. A struct data type lets you encapsulate a set of named members of various types.

IDLos syntax

```
struct AStruct
{
     long    aLongMember;
     short    aShortMember;
};
```

Action language syntax

```
declare AStruct    aStruct;
aStruct.aLongMember = 76543;
```

Example

```
struct Packet
{
    long destination;
    octet data[256];
    boolean retry;
};
```

This defines Packet as a type with members destination, data and retry.

For structs, you can define the types of their members in-line.

```
struct A
{
     boolean firstMember;
     struct B
     {
         unsigned long a;
         long b;
         string c;
     } secondMember;
     long thirdMember;
};
```

General Information

A struct must include at least one member. Because a struct provides a naming scope, member names must be unique only within the enclosing structure.

Recursive definitions of a struct are not allowed. A C-like workaround is available, e.g.,

```
struct nodeList
{
    string name;
    typedef sequence<nodeList> NextNode;
    NextNode nextNode;
};
```

typedef

A typedef lets you define a new legal type; The typedef keyword allows you define a meaningful or simpler name for an IDL type.

IDLos syntax

```
typedef sequence<unsigned long> name;
```

Example

```
typedef enum status {started, finished} racerStatus;
```

General Information

You may rename any legal type. In a typedef, you can define a sequence, struct, or enum in-line.

union

The type union is an IDL composite type. A union type lets you define a structure that can contain only one of several alternative members at any given time. All IDL unions are discriminated.

IDLos syntax

You declare a union type with the following IDL syntax:

```
union name switch (discriminator)
{
    case label1 : element-spec;
    case label2 : element-spec;
    [...]
    case labeln : element-spec;
    [default : element-spec;]
};
```

Action language syntax

```
union MyUnion switch (char)
```

Example

You can access the value of the discriminator in action language using the special _d attribute:

```
union MyUnion switch (char)
{
 case 'S':
 case 's':
     string  aString;
 case 'L':
 case 'l':
     long    aLong;
  default:
     long    justANumber;
};

enum Direction
{
    North,
    South,
    East,
    West,
};
union DirectionUnion switch (Direction)
{
    case South:
    case North:
        long latitude;
    case East:
    case West:
        long longitude;
};
struct ComplexStructure
{
   string name;
   MyUnion myUnion;
};
...
declare MyUnion mu;
mu.aString = ”some data”;
...
if (mu._d == 'S'    mu._d == 's')
{
     // it's a string
}
```

You can also set the discriminator...

```
mu._d = 'L';
```

...but KIS does not check that you are setting the discriminator to a legitimate value.

You can also set the union to null...

```
declare DirectionUnion geoLine;
geoLine = isnull;
```

To handle a union in a complex structure, you must declare a local union to set and then move it as a whole into the structure...

```
declare ComplexStruct myStruct;
declare MyUnion mu;
myStruct.name = “Joe”;
mu.aLong;
```

```
myStruct.myUnion = mu;
```

General Information

The union is stored at runtime as an array of slots for each union member. This storage is not optimized; the runtime size of a union is simply the sum of the union members.

The above means that unions are not the best choice for systems where small size, or high performance are important. Avoid using unions as a placeholder for incomplete analysis; try to use them only when they are the best solution to a problem.

The KIS Monitor displays the discriminator and the current values of all the union members.

If you access an “incorrect” union member (that is, the discriminator indicates another union member is active), the accessor throws an ExceptionDataError system exception.

A struct, union, or exception definition cannot include itself as a member.

A union's discriminator can be integer, char, boolean or enum, or an alias of one of these types; all case label expressions must be compatible with this type. Because a union provides a naming scope, member names must be unique only within the enclosing union. Unions allow multiple case labels for a single member. Unions can optionally contain a default case label. The corresponding member is active if the discriminator value does not correspond to any other label.

unsigned long

The type unsigned long is an IDL simple type. It represents 32-bit unsigned integer values.

IDLos syntax

Inside of an entity block, define an attribute “name” of type unsigned long with:

```
attribute unsigned long name;
```

Define an array “name” of type unsigned long with:

```
typedef unsigned long name[<size>];
```

Define a sequence “name” of type unsigned long with:

```
typedef sequence<unsigned long> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type unsigned long with:

```
declare unsigned long name;
```

Inside of an action language function, allocate an array “name” of type unsigned long with:

```
declare unsigned long name[<size>];
```

Example

```
declare unsigned long name;
```

unsigned long long

The type unsigned long long is an IDL simple type. It represents 64-bit unsigned integer values.

IDLos syntax

Inside of an entity block, define an attribute “name” of type unsigned long long with:

```
attribute unsigned long long name;
```

Define an array attribute “name” of type unsigned long long with:

```
typedef unsigned long long name[<size>];
```

Define a sequence “name” of type unsigned long long with:

```
typedef sequence<unsigned long long> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type unsigned long long with:

```
declare unsigned long long name;
```

Inside of an action language function, allocate an array “name” of type unsigned long long with:

```
declare unsigned long long name[<size>];
```

Example

```
declare unsigned long long name;
```

unsigned short

The type unsigned short is an IDL simple type. It represents 16-bit unsigned integer values.

IDLos syntax

Inside of an entity block, define an attribute “name” of type unsigned short with:

```
attribute unsigned short name;
```

Define an array attribute “name” of type unsigned short with:

```
typedef unsigned short name[<size>];
```

Define a sequence “name” of type unsigned short with:

```
typedef sequence<unsigned short> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type unsigned short with:

```
declare unsigned short name;
```

Inside of an action language function, allocate an array “name” of type unsigned short with:

```
declare unsigned short name[<size>];
```

Example

```
declare unsigned short name;
```

void

The type void is an IDL simple type.

The void keyword is valid only in an operation. In an operation declaration, it may be used to indicate an operation that does not return a function result value.

IDLos syntax

```
void myOperation(in myParm);
```

Example

```
void myOperation(in myParm);
```

wchar

The type wchar is an IDL simple type. It encodes wide characters from any character set. The size of a wchar is platform-dependent.

IDLos syntax

Inside of an entity block, define an attribute “name” of type wchar with:

```
attribute wchar name;
```

Define an array “name” of type wchar with:

```
typedef wchar name[<size>];
```

Define a sequence “name” of type wchar with:

```
typedef sequence<wchar> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type wchar with:

```
declare wchar name;
```

Inside of an action language function, allocate an array “name” of type wchar with:

```
declare wchar name[<size>];
```

Example

```
declare wchar name;
```

wstring

The type wstring is an IDL template type. It is the wide-character equivalent of type string. Like string-types, wstring types can be unbounded or bounded.

Since the KIS implementation of the string type also allows for multi-byte strings, there is little difference between string and wstring. The only current difference is that wstrings cannot be compared as greater-than or less-than other wstrings.

IDLos syntax

Inside of an entity block, define an attribute “name” of type wstring with:

```
attribute wstring name;
```

Define an array “name” of type wstring with:

```
typedef wstring name[<size>];
```

where <size> is an integer and is the number of bytes allocated for the string, (which may not be the number of characters).

Define a sequence “name” of type wstring with:

```
typedef sequence<wstring> name;
```

Action language syntax

Inside of an action language function, allocate a handle “name” of type wstring with:

```
declare wstring name;
```

Inside of an action language function, allocate an array “name” of type wstring with:

```
declare wstring name[<size>];
```

Example

```
declare wstring name;
name = “a single-byte character string”;
```

There is no built in support for wide string literals.

9

IDLos Reference

This chapter contains reference pages for each statement in the Kabira Infrastructure Server (KIS) modeling language, IDLos, in alphabetical order. Each page describes the syntax of the statement, a discussion of usage, and an example.

The page for properties contains a table of all the IDLos properties.

This chapter contains the following sections:

Understanding the notation

The syntax description for each statement or function uses the following conventions:

bold screen font indicates a literal item that appears exactly as shown

italics indicates a language element that is defined elsewhere

`plain text` indicates a feature of the notation, as follows:

(item) parentheses group items together

(item)* the item in parentheses appears zero or more times

(item)+ the item in parentheses appears one or more times

{ item } items in braces are optional

(item1 | item2) the vertical bar indicates either item1 or item2 appears

Common elements of the grammar

The following element of the IDLos grammar appear frequently throughout the reference pages.

Grammar element	Meaning
identifier	any supported characters, digit, or underscore. Must start with a-z or A-Z.
scoped_name	{::} simple_declarator (:: simple_declarator)*
simple_declarator	identifier

Files and engines in IDLos

This section discusses how IDLos can be arranged into files and mapped to KIS engines.

Files

Usually you will put one package in a file. But there is no restriction on how IDLos packages are arranged in files except that each package must be syntactically complete.

If a package name appears again in a model, IDLos reopens the package and adds information to it. In this way, a package might be divided up among many files. For example, you might have each module from that package defined in its own file.

Actions might be defined in separate files. This is similar to the arrangement of C++ implementations kept in different files than the class definitions.

The Design Center does not care how your IDLos files are arranged. Whether you have one file that #includes all the necessary files, or you load each file by hand - the Design Center can build your model as long as you have all the necessary packages loaded when you build your engine.

Engines

You will usually put one package in a file and generate one engine from that package.

Packages are mapped to engines using the KIS Design Center. There is no restriction in the Design Center on how many packages can be built into one engine. You may compile as many packages into an engine as you want.

When you partition KIS applications into different components, the package represents the smallest level of granularity you can use.

action

Defines the behavior of a state or an operation.

Syntax

```
returnValue
action scoped_name ( parameterList )
    raises ( exceptionList )

{‘actionLanguage‘};
```

Description

The implementation of a state or action is defined in action language, between the back quotes ‘ . An action can be defined in an entity, a module, a package, or outside of a package. The action declaration may optional include the signature of the operation or state that it is implementing.

Warnings

None

Example

```
void
action ::Traffic::GUIProxyImpl::updateMode (
   in ModeType modeType,
   in RoadDirection modeDir)
//
{`
    // some action language
`};
```

See also

the section called “ exception” on page 179 , the section called “ stateset” on page 191 .

attribute

An object data member, and its accessors.

Syntax

```
{ readonly } attribute param_type_spec simple_declarator { = literal|scoped_const } ;
```

Description

In an entity, the attribute defines a data member, a set accessor and a get accessor. In an interface, the attribute exposes those same accessors. `readonly` may only be specified in an interface. If the attribute is `readonly` in the interface, only the get accessor is exposed.

Attributes are inherited. To expose the attribute in the interface, it must match the attribute in the entity in both name and type.

Attributes cannot be structs or arrays. However, user-defined types defined as structs or arrays can be attributes.

Attributes may optionally define an initial value. This initial value may be an C/C++ style literal constant or a scoped name denoted by a leading ::

Initial values are only supported for fundamental types, enums and swbuiltin::date.

When an object is created, the values of its attributes without initial values are undefined.

Warnings

None

Example

```
attribute long   timeGreen;
attribute LightColor  colorTarget;
attribute short   baseCount = 16;
attribute string  homeCity = "Peoria";
attribute long   ::mypackage::SomeLongConstant;
attribute swbuiltin::date startingDate = "2006-06-01 09:00:00";
```

See also

the section called “ trigger” on page 193

const

Defines a constant value.

Syntax

```
const const_type identifier = { = literal|scoped_const };
```

Description

literal must be a valid literal type: integer, character, floating point, string, or boolean (see "Literals" on page 12 for more details).

scoped_const is a valid scoped name to another constant definition.

Assigning literal to an identifier of type const_type must represent a valid operation. const_type can be a scoped name. The following table shows valid IDLos types for each literal type:

literal	**const_type**
integer	long long long
character	char wchar
floating point	float double
string	string wstring
boolean	boolean

KIS also performs implicit conversions between string and numeric types in a const statement.

Warnings

A const may not be used as an array index unless it is defined outside the entity or interface in which it appears, for example at the package scope.

Example

```
const long constBoundsValue = 50;
const char littleN = 'n';
const float smallFloat = 73.033e-32;
const string msg1233 = "Don't do that!";
const boolean SureThing = TRUE;
const boolean AnotherSureThing = SureThing;

// used as bound in string, array and sequence
typedef string<constBoundsValue> boundedString;
typedef string stringArray[constBoundsValue];
typedef sequence<string, constBoundsValue> stringSequence;
```

enum

Defines a type whose values are a fixed set of tokens. Optionally defines specific integer values for the tokens.

Syntax

```
enum identifier {identifier (= value(, identifier (= value)))*};
```

Description

Enums contain a comma-separated list of one or more enumerators, which are just identifiers. These identifiers may optionally contain an assigned value, which may be either a literal constant integer value (a signed 32 quantity) or another scoped identifier.

Warnings

None

Example

```
enum ModeType
{
    tyTimed,
    tySensed,
    tyHoldGreen = 77,
    tyBlinkYellow,
    tyBlinkRed = lights::Red,
    tyNoChange
};
```

entity

Defines the implementations of objects.

Syntax

```
entity simple_declarator {: scoped_name} {(entityContent)*};
entity simple_declarator;
```

Description

The first syntax is for entity definition, the second for forward declaration.

Entities can contain actions, attributes, enums, exceptions, keys, operations, signals, transitions, statesets, structs, triggers, and typedefs.

An entity statement may have extentless, managedextent, annotation, local, and singleton properties.

To make an entity accessible outside its package, expose the entity with an interface. An entity may be exposed by more than one interface.

The : scoped_name following the entity declarator designates a super type for inheritance. Entities may inherit from other entities if they are in the same package, and have the same value of their `local` property.

Entities may have relationships to other entities in the same package.

Forward declares allow you to use an entity type before it is defined.

Warnings

Local entities have such a different description, they are covered on their own page.

An entity must be defined, not just forwarded, before it can be inherited from.

Multiple inheritance is not supported.

Example

```
// forward declares
entity TimerEventImpl;

// definitions
entity TimerEventImpl
    {
    attribute Road road;
    oneway void generate ();
};
```

See also

the section called “ local entity” on page 184 , the section called “ trigger” on page 193

exception

A user-defined type that can be thrown and caught upon error.

Syntax

```
exception identifier {: scoped_name} { (type_spec declarator;)*};
```

Description

Define exceptions when you need to report errors in your applications.

The scoped_name following the exception declarator designates a super type for inheritance. Exceptions may inherit from other exceptions.

Warnings

None

Example

```
exception ExpFileNotFound
{
    string path;
    string filename;
};
```

See also

the section called “operation” on page 187

action language “Exceptions” on page 163

expose

Specifies which entity an interface exposes.

Syntax

```
expose entity scoped_name with interface scoped_name;
```

Description

An expose statement may have the `annotation` property. An abstract interface may not be used in an expose statement. You may define expose statements in a package or module.

Warnings

None

Example

```
expose entity GuiProxyImpl with interface GuiProxy;
```

See also

the section called “ interface” on page 180 , the section called “ entity” on page 178

interface

Interfaces provide access to entities for clients in other packages.

Syntax

```
interface simple_declarator {: scoped_name} {(interface_body)*};
interface simple_declarator;
```

Description

The first syntax is for interface definition. The second syntax is for forward declaration.

Interfaces can contain attributes, enums, exceptions, operations, structs, and typedefs.

Interfaces define what part of an entity gets exposed outside of the package boundary. Everything has open access within a package.

Interfaces contain the attributes and operations of the entity that you intend to expose to outside clients.

Signals are exposed as oneway void operations. They must be implemented in the entity.

The : scoped_name following the interface declarator designates a super type for inheritance. Interfaces may inherit from other interfaces, either within the same package or across package boundaries.

An `interface` statement may have `abstract` , `annotation` , `createaccess` , `deleteaccess` , `extentless` , `extentaccess` , managedextent and `singleton` properties.

You may also use interfaces for access to the entity from within the package; but it is not necessary.

Warning

Relationships defined between interfaces are implemented between the entities they expose. Interfaces may not participate in associative relationships.

All attributes and operations listed in an interface must be defined in the exposed entity.

An interface must be defined, not just forwarded, before it can be inherited from.

Example

```
interface Employee
{
// only these parts of EmployeeImpl will be
// accessible outside the package.
    void promote();
    oneway void makewaves();
    attribute string name;
};

entity EmployeeImpl
{
    void promote();
    attribute string name;
    attribute string m_nickname;
    signal makewaves();
};
expose entity EmployeeImpl with interface Employee;
```

See also

the section called “ entity” on page 178 , the section called “ expose” on page 180

key

A list of attributes whose values are used to select one or more instances of an entity.

Syntax

```
key simple_declarator  {simple_declarator  (, simple_declarator)*};
```

Description

Each key is a named list of attributes.

A key can contain as many attributes as you want. Multiple keys may be defined for a single entity.

Keys can be defined on both entities and roles in relationships. Keys defined on entities index all instances in the entity extent. Keys defined on roles only index the instances currently in the role.

Attempting to create an object with key values that are already part of a unique index will cause the runtime to throw a swbuiltin::ExceptionObjectNotUnique exception.

A `key` statement may have the `annotation, unique, or ordered` properties.

Warnings

Arrays and sequences are not valid in keys.

When a key is inherited (that is, if an entity containing a key has subtypes), the index for the key is maintained in the type where the key is defined. This means that keys must be unique across instances of that type and all its subtypes.

Avoid modifying key attributes. If it is necessary to modify a attribute which is part of a key, enclose the modifications in a try..catch() block. This allows KIS to update the keys immediately after they are modified, and it lets you catch the ExceptionObjectNotUnique exception that can be thrown when you change a key. If you don't catch this exception, the engine will crash.

Here is an example of how you might use a temporary object when you need to change an attribute which is part of a key:

```
entity KeyedEntity
{
    attribute long longAttr;
    key Primary { longAttr };
};

. . . . . IDLos above, action language below . . . . .

declare KeyedEntity  keyedEntity;
declare long orgKey = 3;
declare long newKey = 4;

//
//  Always used a values clause when creating a keyed entity
//
create keyedEntity values (longAttr:orgKey);

//
//  Change the key
//
try
{
  keyedEntity.longAttr = newKey;
}
catch (swbuiltin::ExceptionObjectNotUnique)
{
  keyedEntity.longAttr = orgKey;
}
```

Key attributes should always be modified as a group within a try block.

If a key does not need to be modified after an object has been created the entity attribute (and any associated interface attribute) should be marked readonly. This allows the runtime to significantly optimize queries against that key value since it does not need to worry about key changes occurring once the index is created for an object. For example:

```
interface MyObject
{
     readonly attribute long id;

     key K { id };
};
entity MyObjectImpl
{
     readonly attribute long id;

     key K { id };
};
expose entity MyObjectImpl with interface MyObject;
```

The IDLos auditor will detect any attempts to modify this key at build time and fail to build the model.

Example

```
entity Employee
{
    attribute string firstName;
    attribute string lastName;
    attribute long SSN;
    attribute long employeeNumber;
    key primary {employeeNumber};
    key secondary {SSN};
};


entity E
{
   attribute string s;
};

entity X
{
   attribute long l;
};

relationship RUnique
{
   role E has 0..* X
   {
      key K { l };
   };
   role X back 0..* E
   {
      key K { s };
   };
};
};
```

See also

the section called “ attribute” on page 175

action language `select` , `for ... in`

local entity

An entity that cannot be distributed. A local entity is essentially a C++ class defined in IDLos and implemented in action language. Its operations are invoked directly without being dispatched through the KIS event bus, so they execute somewhat faster.

Though their operations are executed in a transaction, local entities are not transactional. This means that the invocation of operations in a local entity is not logged in the transaction log, and changes to the local entity are not logged in the transaction log, but everything that the local entity does to other entities is logged.

You specify a local entity by placing the `local` property before the entity declaration.

Syntax

```
[local] entity simple_declarator {: scoped_name} {(entityContent)*};
entity simple_declarator;
```

The first syntax is for entity definition, the second for forward declaration.

Description

Local entities can contain actions, enums, exceptions, operations, structs, and typedefs. To make a local entity accessible outside its package, expose the entity with an interface. A local entity may:

- be exposed by more than one interface
- inherit from other local entities in the same package
- use the `annotation` property
- have operations that use the `packageinitialize, packagereload, initialize , unload, reload` , and `terminate` properties

Local entities may not have relationships to other entities. Forward declares allow you to use a local entity type before it is defined.

Do not explicitly create or delete local entities. Declaring a local entity is sufficient to instantiate it.

Warnings

A local entity cannot be used as an actual parameter when calling an operation on a non-local entity.

Local entities are not transactional.

Local entities have no state, so they cannot contain statesets.

Example

```
// forward declare
entity StartUp;

[local]
entity StartUp
{
    [initialize]
    void initData ();
};
```

See also

the section called “ entity” on page 178

module

Used to provide extra name spaces within a package as needed.

Syntax

```
module simple_declarator {(definition)+};
```

Descriptions

Modules define a name space. Modules may be nested.

A module may contain entities, interfaces, enums, structs, exceptions, typedefs, nested modules, relationships, actions, and exposes statements. Modules are declared inside of packages.

A `module` statement may have the `annotation` property.

When a module name appears more than once in a model, information is added to the module definition each subsequent time that its name appears.

Warnings

None

Example

```
package Traffic
{
    module Pedestrians
    {
        entity Joggers {};
    };
    module Cars
    {
```

```
        entity Trucks {};
    };
    relationship
    {
        role Pedestrians::Joggers avoid 0..* Cars::Trucks;
        role Cars::Trucks stop 0..* Pedestrians::Joggers;
    };
};
```

native

Used to alias a C or C++ type as an IDLos type

Syntax

```
native {scoped_name} simple_declarator;
```

Descriptions

Native is used to map an external type into the IDLos type system.

The optional scoped_name is used to specify the C or C++ type that should be mapped to the IDLos type name specified in simple_declarator.

The native type is defined in models using the simple_declarator name.

Warnings

Native types are not marshaled over distribution. The value of a native type on a remote node is undefined.

Exposing native types through an interface is strongly discouraged. Native types are usually only valid in a single process. Exposing them in an interface makes the type potentially visible across engine boundaries.

Example

```
native ::SWSMOffset     Offset;
native ::FILE     *FilePtr;
native ::CGObject     *ObjPtr;
typedef ObjPtr     ObjPtrTypeDef;

entity A
{
  attribute FilePtr    filePtr;
  attribute ObjPtr    objPtr;
  attribute ObjPtrTypeDef    objPtrTypeDef;
  attribute Offset    offset;
};

//
//   Using them in action language
//
a.filePtr = ::fopen("foo", "a");
if (::fclose(a.filePtr) != 0)
{
```

```
  printf("FAILED: fclose failed %d\n", errno);
  SWBuiltIn::stop(1);
}

declare ObjPtr objPtr;

a.objPtr = new ::CGObject;
objPtr = a.objPtr;
delete objPtr;
a.objPtr = NULL;

a.objPtrTypeDef = new ::CGObject;
objPtr = a.objPtrTypeDef;
delete objPtr;
a.objPtrTypeDef = NULL;
```

operation

Declares an operation on an entity or an interface.

Syntax

```
{oneway}(param_type_spec   void) simple_declarator
                               ( { param_dcl (, param_dcl)*} )
            {raises (scoped_name (, scoped_name)*)};

param_dcl:     (in   out   inout) param_type_spec simple_declarator
```

Description

Use oneway for asynchronous operations.

In an entity, an operation may have `annotation` , `const` , `inline, local` , `pure` , and `virtual` properties. The `pure` property may only be used together with `virtual` .

The inline property can only be used on operations that are not exposed by an interface, are not declared virtual, and are not used outside of the entity that defines the operation.

In an interface, an operation may have `annotation` , `const` , and `local` properties.

In a local entity, an operation may have `annotation` , `const` , `packageinitialize, packagereload, initialize` , `unload, load` , and `terminate` properties.

Operations are implemented in `action` statements.

Warnings

Operations have constraints on whether they can be oneway, twoway, or have parameters when they:

- are used in triggers
- are used as engine events
- implement exposed signals

See the appropriate sections of the manual.

Example

```
void updateMode (
    in ModeType modeType,
    in RoadDirection modeDir);
void updateLightTimes (
    in long timeGNS,
    in long timeGEW,
    in long timeYNS,
    in long timeYEW);
void updateSensor (
    in boolean newCar,
    in SideDirection carDir);
LightColor getLightN ();
LightColor getLightS ();
LightColor getLightE ();
LightColor getLightW ();
void getAllLights (
    out LightColor north,
    out LightColor south,
    out LightColor west,
    out LightColor east);
```

See also

the section called “ action” on page 175 , the section called “ signal” on page 190 , the section called “ trigger” on page 193

package

Packages define components.

Syntax

```
package simple_declarator { (definition) * };
```

Description

IDLos applications require packages. Packages define components. They form an access boundary: you cannot access an entity in another package unless it is exposed by an interface.

A package may contain entities, interfaces, enums, structs, exceptions, typedefs, modules, relationships, actions, and exposes statements.

A `package` statement may have the `annotation` property.

When a package name appears more than once in a model, information is added to the package definition each subsequent time that its name appears.

An IDLos file may contain more than one package, and a package may be contained in more than one file.

Warnings

There is no implicit package scope for a file. All IDLos statements except `action` definitions must be encapsulated in an explicit package.

Example

This fragment of the sample model has been edited for clarity's sake.

```
package Traffic
{
    enum LightColor
    {
        tyRed,
        tyYellow,
        tyGreen,
        tyBlkRed,
        tyBlkYellow
    };
    interface GUIProxy
    {
        // operations
        LightColor GetLightN ();
        LightColor GetLightS ();
        LightColor GetLightE ();
        LightColor GetLightW ();
    };
};
```

relationship / role

Defines a relationship between entities or between interfaces.

Syntax

```
relationship simple_declarator
{
   role scoped_name identifier(0..1   0..*   1..1
                                        1..*) scoped_name;
  {role scoped_name identifier(0..1   0..*   1..1
                                      1..*) scoped_name;}
 {using scoped_name;}
};
```

Description

You specify roles in one or both directions between entities in the relationship statement. Each role has a name, a multiplicity, and specifies the two related entities. You can also specify an associative entity with the optional `using` clause.

You can specify a relationship between interfaces instead of between entities. This is how you expose a relationship. Do not mix entities and interfaces in a relationship. Interfaces may not be used as associative objects; relationships between interfaces may not contain a `using` clause.

A `relationship` statement may have the `annotation` property.

Warnings

The two roles in a relationship must have different names.

Example

```
relationship flowOwnership
{

 role intersection has 1..1 trafficFlow;
 role trafficFlow isin 1..1 intersection;
};
```

See also

the section called “ trigger” on page 193

action language `relate` and `unrelate`

signal

Defines an event for a state machine.

Syntax

```
signal simple_declarator (param_dcl (, param_dcl)*);

param_dcl:
in param_type_spec simple_declarator
```

Description

A `signal` statement may have the `annotation` property.

Warnings

Signals may have only `in` parameters.

Example

```
signal blinkRed( );
signal goRed( );
signal blinkYellow( );
signal greenTimed( );
signal greenSensor( );
signal greenHold( );
signal timeout( );
signal carArrived( );
signal goYellow( );
```

See also

the section called “ stateset” on page 191 , the section called “ transition” on page 192 , the section called “operation” on page 187

stateset

Defines the allowed states for an entity.

Syntax

```
stateset {simple_declarator (, simple_declarator)*} = simple_declarator;
```

Where the last simple_declarator designates the initial state.

Description

Defines the states in an entity’s state machine, and its initial state. When the entity is created, it is placed into its initial state.

Each state (including initial and final states) must be associated with an `action` . In KIS, the action of an initial state is not executed upon object creation. You must always transition into a state with a signal in order for an action to be executed.

A stateset statement may have `annotation` and `finished` properties.

Warnings

State names are in the entity’s name space.

Example

```
stateset
{
    Initial,
    BlinkRed,
    Red,
    TimedGreen,
    TimedYellow,
    SensorGreenPreferred,
    WaitForCar,
    WaitForGreenTimeout
} = Initial;
```

See also

the section called “ action” on page 175 , the section called “ signal” on page 190 , the section called “ transition” on page 192

struct

A type to hold data.

Syntax

```
struct identifier {: scoped_name} { (type_spec declarator;)*};
```

Description

Use structs as a container for data.

The scoped_name following the struct declarator designates a super type for inheritance. Structures may inherit from other structures.

Warnings

None

Example

```
struct Result
{
    typedef sequence<string> Messages;
    Messages messages;
    boolean status;
};
```

transition

Defines the transition from one state to another when a specific signal is received.

Syntax

```
transition scoped_name to (scoped_name   cannothappen   ignore)
          upon simple_declarator;
```

Description

Transition statements specify how a state machine should respond to a certain signal when the entity is in a particular state. You can transition to another state, ignore the signal, or cause an exception to be thrown by specifying `cannothappen` . If a transition is not specified, the default is `cannothappen` .

Warnings

There are no automatic transitions in IDLos. All transitions must be explicit.

Example

```
transition Initial to BlinkRed upon blinkRed;
transition BlinkRed to Red upon goRed;
transition Red to BlinkRed upon blinkRed;
transition Red to BlinkYellow upon blinkYellow;
transition Red to TimedGreen upon greenTimed;
transition Red to SensorGreenPreferred upon greenSensor;
transition TimedYellow to Red upon timeout;
transition CarWaiting to ignore upon carArrived;
transition CarWaiting to ignore upon goYellow;
transition HoldGreen to ignore upon carArrived;
transition WaitForGreenTimeout to ignore upon carArrived;
```

See also

the section called “ signal” on page 190 , the section called “ stateset” on page 191

trigger

Triggers invoke operations upon the occurrence of some event.

Syntax

```
trigger simple_declarator upon (create   delete   update | refresh
                                commit   abort);
trigger simple_declarator upon (pre-get   post-get   pre-set
                                post-set) simple_declarator;
trigger scoped_name upon ( relate   unrelate ) simple_declarator;
```

Description

The first syntax is for entity triggers: The simple_declarator specifies an operation name. You define entity triggers in entities.

The second syntax is for attribute triggers. The first simple_declarator specifies an operation or signal. The next simple_declarator specifies the attribute upon which to trigger. These are also defined in entities.

The third syntax is for role triggers. The scoped_name identifies an operation or signal in the from entity of the role (the first one mentioned in the `role` statement). The simple_declarator specifies the `role` name. Role triggers are defined within relationship statements.

A `trigger` statement may have the `annotation` property.

Create, update, and delete triggers with inheritance If you define create or update triggers for both the super type (parent) and subtype (child), then the super type trigger fires before the one in the subtype. Conversely, subtype delete triggers fire before those in a super type. Some readers will find this familiar, as it resembles class constructor/destructor behavior in other languages.

Warnings

Operations used in triggers must return `void` and must have no parameters, and delete triggers may not be one-way operations. The operations must be defined in the entity for which the triggers are defined or in a super type.

Example

```
trigger initializeAttributes upon create;
trigger calculatePayment upon pre-get mortgagePayment;
trigger deleteCustomer upon unrelate patronizes;
```

See also

the section called " attribute" on page 175 , the section called "operation" on page 187 , the section called " signal" on page 190 , the section called " entity" on page 178 , the section called " relationship / role" on page 189

typedef

Give a new name to a type. Define a new sequence or array type.

Syntax

```
typedef type_spec declarator;
```

Description

The declarator is the name of the new legal type. A declarator can include array dimensions.

The type_spec can be any legal type already defined, either a basic type, or a user-defined type. It could also be a sequence or a bounded string.

Warnings

None

Example

```
typedef octet byte;
typedef sequence<Employee> Division;
typedef boolean low_pass_filter[3][3];
```

Complete IDLos grammar

```
accessValues  :

   granted     revoked
```

```
actionStatement  :
  action  scoped_name  swalBlock  ;
annotation  :
  annotation  =  string_literal  (  string_literal  )*
annotations  :
  [  annotation  ]
any_type  :
  any
array_declarator  :
  quoted_identifier  (  fixed_array_size  )+
assignedProperty  :
 annotation     finished  =  finishedStateList
 createaccess  =  accessValues    deleteaccess  =  accessValues
 extentaccess  =  accessValues  | sizehint = integer_type |
 lockmode = lockMode

assocSpec  :
  using  scoped_name

attr_dcl  :
  {  readonly  }  attribute  param_type_spec  simple_declarator
  {  =  string_literal  }
  (  ,  simple_declarator  {  =  string_literal  }  )*

attributeList  :
  {  simple_declarator  (  ,  simple_declarator  )*  }

attributeTrigger  :
  (  pre-set     post-set     pre-get     post-get  )
    simple_declarator

base_type_spec  :
  floating_pt_type     integer_type     char_type     wide_char_type
  boolean_type    octet_type     any_type     object_type
```

```
booleanProperty  :
  abstract  |  const  |  dynamiclog | extentless | initialize |
  inline | local  |  managedextent  | ordered |
  singleton  |  terminate  |  packageinitialize  |
  packagereload | pure  |  unique |
  unload | virtual

boolean_literal  :
  TRUE   true     FALSE   false


boolean_type  :
  boolean


case_clause  :
  (  case_label  )+  element_spec  ;


case_const_exp  :
  const_exp     unquoted_scoped_name


case_label  :
  case  case_const_exp  :      default  :


char_type  :
  char


character_literal  :
  character-literal


complex_declarator  :
  array_declarator


const_dcl  :
  const  const_type  simple_declarator  =  const_exp
```

```
const_exp  :
  {  (  -     +  )  }  literal

const_type  :
  integer_type     char_type     octet_type     wide_char_type
  boolean_type     floating_pt_type     string_type     wide_string_type
  fixed_pt_const_type    scoped_name

constr_type_spec  :
  struct_type     union_type     enum_type



context_expr  :
  context  (  string_literal  (  ,  string_literal  )*  )

declarator  :
  simple_declarator     complex_declarator

declarators  :
  declarator  (  ,  declarator  )*

definition  :
  type_dcl  ;     const_dcl  ;     except_dcl  ;     interface  ;
  module  ;     entity  ;    exposeStatement  ;
  relationshipStatement  ;     actionStatement     includeStatement
  pragmaStatement

element_spec  :
  type_spec  declarator

entity  :
  entity  (  entityStatement     forward_entity_dcl  )
```

```
entityBlock  :
  {  (  {  properties  }  entityContent  )*  }

entityContent  :
  signal_dcl  ;     outerStateSetStatement  ;
  transitionStatement  ;    entityTriggerStatement  ;
  keyStatement  ;     type_dcl  ;     const_dcl  ;     except_dcl  ;
  attr_dcl  ;     op_dcl  ;     actionStatement

entityStatement  :
  simple_declarator  {  inheritance_spec  }  entityBlock

entityTrigger  :
  (  create     delete     refresh     abort     commit
    state-conflict | update  )

entityTriggerStatement  :
  trigger  simple_declarator  upon
  (  attributeTrigger     entityTrigger  )

enum_type  :
  enum  simple_declarator  {  enumerator  (  ,  enumerator  )*  }

enumerator  :
  simple_declarator

except_dcl  :
  exception  simple_declarator  {  inheritance_spec  } {  (  except_member  )*  }

except_member  :
  type_spec  declarators  ;

exposeStatement  :
  expose  ( entity  scoped_name  with  interface  scoped_name  )
```

```
finishedStateList  :
  {  stateName  (  ,  stateName  )*  }

fixed_array_size  :
  [  template_bounds  ]

fixed_pt_const_type  :
  fixed

fixed_pt_type  :
  fixed  <  positive_int_const  ,  integer_literal  >

floating_point_literal  :
  floating-point-literal

floating_pt_type  :
  float     double     long  double

forward_dcl  :
  simple_declarator

forward_entity_dcl  :
  simple_declarator

includeStatement  :
  include  filename

inheritance_spec  :
  :  scoped_name  (  ,  scoped_name  )*

initState  :
  stateName
```

```
integer_literal  :
  decimal-integer-literal     hexadecimal-integer-literal     octal-integer-literal

integer_type  :
  signed_int     unsigned_int

interface  :
  interface  (  interface_dcl     forward_dcl  )

interfaceContent  :
  type_dcl  ;     const_dcl  ;     except_dcl  ;     attr_dcl  ;
  op_dcl  ;     keyStatement  ;

interface_body  :
  (  {  properties  }  interfaceContent  )*

interface_dcl  :
  simple_declarator  {  inheritance_spec  }  {  interface_body  }

keyStatement  :
  key  simple_declarator  attributeList

literal  :
  integer_literal     floating_point_literal     boolean_literal
  string_literal    character_literal

lockMode  :
  normal     dirtyread

member  :
  type_spec  declarators  ;

member_list  :
  (  member  )+
```

```
module  :
  module  simple_declarator  {  (  {  properties  }  definition  )+  }

native_type  :
  native  (  scoped_name  (  *  )*  )* simple_declarator

object_type  :
  Object

octet_type  :
  octet

op_attribute  :
  oneway
op_dcl  :
  {  op_attribute  }  op_type_spec  simple_declarator  parameter_dcls
    {  raises_expr  } {  context_expr  }

op_type_spec  :
  param_type_spec     void

outerStateSetStatement  :
  stateSetSpec  =  initState

packageBlock  :
  {  (  {  properties  }  definition  )*  }

packageScope  :
  simple_declarator  ::

packageStatement  :
  package  simple_declarator  packageBlock  ;
```

```
param_attribute  :
  in     out     inout

param_dcl  :
  param_attribute  param_type_spec  simple_declarator

param_type_spec  :
  base_type_spec     string_type     wide_string_type     fixed_pt_type
  scoped_name

parameter_dcls  :
  (  {  param_dcl  (  ,  param_dcl  )*  }  )

positive_int_const  :
  integer_literal

pragmaStatement  :
  pragma  include filename

properties  :
  [  property  (  ,  property  )*  ]

property  :
  assignedProperty     booleanProperty


qualifiedName  :
  simple_declarator  (  .  simple_declarator  )*

quoted_identifier  :
  identifier     string-literal

raises_expr  :
```

```
  raises  (  scoped_name  (  ,  scoped_name  )*  )

relationshipStatement  :
  relationship  simple_declarator
  {  roleStatement  ;  (  roleTriggerStatement  ;  )*
    {  roleStatement  ;  (  roleTriggerStatement  ;  )*
       {  assocSpec  ;  (  roleTriggerStatement  ;  )*  }
     }
  }

roleMult  :
  1..1     0..1     1..*     0..*

roleStatement  :
  {  annotations  }  role  scoped_name simple_declarator
                   roleMult  scoped_name { keyStatement  ; }

roleTriggerStatement  :
  trigger  scoped_name  upon  (  relate     unrelate  )
           simple_declarator

scoped_name  :
  {  ::  }  simple_declarator  (  ::  simple_declarator  )*

sequence_type  :
  sequence  <  simple_type_spec  {  ,  template_bounds  }  >

signal_dcl  :
  signal  simple_declarator  parameter_dcls

signed_int  :
  signed_long_int     signed_short_int     signed_longlong_int
```

```
signed_long_int  :
  long


signed_longlong_int  :
  long  long


signed_short_int  :
  short


simple_declarator  :
  quoted_identifier


simple_type_spec  :
  base_type_spec     template_type_spec     scoped_name


socleContent  :
  packageStatement     actionStatement     cppStatement


socleStatement  :
  (  {  properties  }  socleContent  )+


stateName  :
  simple_declarator


stateSetSpec  :
  stateset  {  stateName  (  ,  stateName  )*  }


string_literal  :
  string-literal


string_type  :
  string  {  <  template_bounds  >  }


struct_type  :
```

```
  struct  simple_declarator  {  inheritance_spec  } {  member_list  }


swalBlock  :
  {  delimited-action-language  }


switch_body  :
  (  case_clause  )+


switch_type_spec  :
  integer_type     char_type     boolean_type     enum_type     scoped_name


template_bounds  :
  positive_int_const     scoped_name


template_type_spec  :
  sequence_type     string_type     wide_string_type     fixed_pt_type


transitionStatement  :
  transition  qualifiedName  to  transitionTarget
                              upon  simple_declarator

transitionTarget  :
  qualifiedName     cannothappen     ignore


type_dcl  :
  typedef  type_declarator     struct_type     union_type     enum_type
            native_type


type_declarator  :
  type_spec  declarators


type_spec  :
  simple_type_spec     constr_type_spec
```

```
union_type  :
  union  simple_declarator  switch  (  switch_type_spec  )
        {  switch_body  }

unquoted_scoped_name  :
  {  ::  }  identifier  (  ::  identifier  )*

unsigned_int  :
  unsigned  (  unsigned_short_int     unsigned_long_int
                unsigned_longlong_int  )

unsigned_long_int  :
  long

unsigned_longlong_int  :
  long  long

unsigned_short_int  :
  short

wide_char_type  :
  wchar

wide_string_type  :
  wstring  {  <  template_bounds  >  }
```

10

Action language reference

This chapter provides a reference to the Kabira Infrastructure Server (KIS) action language, organized by keyword. Each statement or function in the action language appears on its own page, with a top-level syntax definition, a description of the statement or function, and an example. See the section called "Complete action language grammar" on page 243 for a complete syntactic definition of the action language.

This chapter contains the following sections:

- the section called " assert" on page 209, the section called " break" on page 210, the section called "cardinality" on page 211, the section called " continue" on page 212, the section called " create" on page 213
- the section called " create singleton" on page 214, the section called "declare" on page 215, the section called " delete" on page 217
- the section called " empty" on page 218, the section called " Exceptions" on page 219, the section called " for" on page 222, the section called " for ... in" on page 223
- the section called "if else else if" on page 224, the section called "in" on page 225, the section called "isnull" on page 226, the section called "relate" on page 227, the section called "return" on page 228
- the section called "select" on page 229, the section called "select...using" on page 231, the section called "self" on page 232, the section called "spawn" on page 233
- the section called " Transactions" on page 238, the section called " Types" on page 240, the section called " unrelate" on page 241, the section called "while" on page 242
- the section called "Complete action language grammar" on page 243

About the notation

The syntax description for each statement or function uses the following conventions:

bold screen font indicates a literal item that appears exactly as shown

italics indicates a language element that is defined elsewhere

`plain text` indicates a feature of the notation, as follows:

(item) parentheses group items together

(item)* the item in parentheses appears zero or more times

(item)+ the item in parentheses appears one or more times

{ item } items in braces are optional

(item1 | item2) the vertical bar indicates either item1 or item2 appears

For example:

```
keyword { optionalKeyword }
     { optionalThing }( zeroOrMoreRepeatingThing )*
     ( oneAlternative | otherAlternative | thirdAlternative )
```

Some common elements

The following paragraphs describe some common elements found in the action language grammar. These are not statements or functions, so they do not appear under their own headings later in this chapter. Nor does the action language grammar (see the section called “Complete action language grammar” on page 243) explain their meaning.

handle

A handle is a reference to an object. When initially declared, a handle is empty and may be used only as the target of an assignment, select, or create statement.

chainSpec

A chainSpec represents a relationship traversal. When one entity or interface has a relationship to another entity or interface, two objects may be related using a relate statement. The relationship can then be traversed from one object to the other; this involves a chain spec of the form:

scopedType [roleName]

For example, the for..in statement:

for handle in setSpec { where whereSpec } statementBlock

accepts either a chainSpec or an entity name for setSpec. The chainSpec might be used as in either of the following examples:

```
// verify this subscriber's phone services
for s in thisSubscriber->Service[subscribesTo]
{
    service.verify();
}

// reorder all the parts used in this assembly
for p in thisAssembly->Component[contains]->Part[includes]
```

```
{
    part.reorder();
}
```

A chainSpec may also represent an associative relationship traversal. An associative relationship is where one entity has a relationship with another. This relationship can only be traversed using the third entity. For example:

```
select thisSeller from Seller where thisSeller.name = "fred";

for aBuyer in thisSeller->Sale[sell_to]->Buyer[sell_to]

{

   aBuyer.verifyCredit();

}
```

assert

Model level assertion.

Syntax

```
assert ( expression );
```

Description

expression can be any valid boolean expression defined in IDLos.

When an assertion fails, the runtime logs an error message as a fatal error in the trace file and sends the error message to `stderr` . The message includes the model file name and line number where the assertion failed.

The following is an example of an assertion message:

```
x.soc:59: failed assertion '(idx > 1)'
```

After the runtime logs the message, it throws a runtime exception in the model.

IDLos constraints

None.

Exceptions

None.

Warnings

Assertions are enabled only for DEVELOPMENT builds; they do not appear in PRODUCTION builds.

Examples

```
assert( !empty myobj );
assert( idx >= 1 );
assert( myEnum == One    myEnum == Two );
assert( self.boolOp() == true );
```

See also

break

Exit from a while or for loop.

Syntax

```
break;
```

Description

This loop control statement lets you exit immediately from a while, for...in, or for loop. If while or for loops are nested, the break exits from the containing loop only.

IDLos constraints

None.

Exceptions

None.

Warnings

None.

Example

```
declare long x;
for (x = 0; x < 10; x++)
{
    //
    //    Check for termination condition.  If
    //    found get out of loop immediately
    //
    if (someCondition)
    {
        break;
    }
}
```

See also

the section called “ continue” on page 212 , the section called “ for” on page 222 , the section called “ for ... in” on page 223 , the section called “return” on page 228 , the section called “while” on page 242

cardinality

Return the number of objects in an extent or relationship.

Syntax

```
cardinality ( ( className   (handle   self) -> chainSpec ) );
```

Description

cardinality returns the number of objects in an extent or relationship. Cardinality can be used on extents and a single relationship chain. Extended relationship chains are not allowed.

cardinality should be used instead of iterating over an extent or relationship to determine the number of objects. This has a large performance advantage over iteration since extents and relationships maintain the number of objects directly, eliminating the requirement to iterate over the entire extent or relationship.

IDLos constraints

None

Exceptions

None.

Warnings

When you use cardinality on extentless entities, the result is approximate.

Example

```
entity E { };
entity I { };

relationship EToI
{
    role E toI 1..1 I;
    role I toE 1..* E;
};

// --- Get the number of instances of E (the extent) ---
declare long numE = cardinality (E);

// --- Get the number of instances of E related to I ---
declare long numIToE = 0;
declare I    anI;
```

```
// Iterate over the extent adding up the number of
// related instances of E
for anI in I
{
    numIToE = numIToE + cardinality (anI->E[toE]);
}
```

See also

the section called " for ... in" on page 223 , the section called "in" on page 225 ,

continue

Skip immediately to the next iteration of a for, for...in, or while statement.

Syntax

```
continue;
```

Description

This loop control statement skips immediately to the next iteration of a for, for...in, or while statement.

IDLos constraints

None.

Exceptions

None.

Warnings

None.

Example

```
declare Customers aCustomer;
for aCustomer in Customers
{
    //    Check to see if we have already processed
    //    this customer.  If we have, skip this customer
    //    and proceed to the next

    if (aCustomer.seen)
    {
        continue;
    }

    //    First time we have seen this customer.  Welcome
    //    them.
}
```

See also

the section called " break" on page 210 , the section called " for" on page 222 , the section called " for ... in" on page 223 , the section called "return" on page 228 , the section called "while" on page 242

create

Create an object with initial values.

Syntax

```
create handle { values ( valueSpec (, valueSpec )* ) } ;
```

Description

This statement creates an object using a previously declared handle of entity or interface type. A handle is set to empty upon declaration, and is invalid (that is, it does not refer to any object) until it is assigned to a valid object or used in a create statement.

valueSpec is an initial value assignment for the attributes or relationships, in the form attribute-name:value (for relationships, chainSpec:value) If attributes are not assigned initial values, the value of the attribute is undefined. The only exception to this is object references. Object references are initialized to a value of empty.

IDLos constraints

The entity being created cannot have been declared as a singleton.

Exceptions

```
swbuiltin::ExceptionObjectNotUnique
```

A duplicate key was detected. Update the values statement to contain unique values for all keys in the object.

```
swbuiltin::ExceptionResourceUnavailable
```

A resource required to create the object was unavailable. This can happen if this is a create of a remote or a persistent object.

Warnings

It is an error to use create on entities that have been declared as singleton in your model. You must use the section called " create singleton" on page 214 instead.

Attributes that are part of a key must be initialized to a unique key value to avoid duplicate key exceptions during object creation.

Relationships that are used for referential integrity in a backing database should be initialized during creation to ensure that referential integrity is maintained.

Examples

```
entity Customer
{
    attribute string   name;
    attribute long     id;
    key CustId { id };
};

declare Customer  lclCustomer;
declare Customer  rmtCustomer;
declare long      uniqueKey;

//
// allocateCustId allocates a unique customer id
//
uniqueKey = self.allocateCustId();

//
// Create a customer on the local node
//
create lclCustomer values (name:"Joe", id:uniqueKey);

//
// Allocate another customer id
//
uniqueKey = self.allocateCustId();

//
//    Create a customer
//
create rmt Customer values (name:"Jean-Louis", id:uniqueKey);
```

See also

the section called " create singleton" on page 214 , the section called " delete" on page 217 ,

create singleton

Create a singleton object with initial values.

Syntax

```
create singleton  handle { values ( valueSpec (, valueSpec )* ) };
```

Description

This statement creates a singleton object using a previously declared handle of entity or interface type. There is only one instance of a singleton entity.

create singleton can be called multiple times. If the single instance has not been created it is created and returned to the caller. If the instance already exists, a handle to the single instance is returned to the caller.

The initial values specified are only applied if the singleton is created. If the singleton was created by a previous create singleton, the initial values have no affect.

IDLos constraints

Entity must be declared as a singleton.

Exceptions

```
swbuiltin::ExceptionResourceUnavailable
```

A resource required to create the object was unavailable. This can happen if this is a create of a remote singleton.

Warnings

It is an error to use create singleton on entities and interfaces that have not been declared as a singleton in IDLos.

Initial values are only applied when the singleton is created.

Example

```
[ singleton ]
entity PortAllocator
{
    void doInitialization();
};

//
// Create a singleton
//
declare PortAllocator myAllocator;

create singleton myAllocator;

//
//        Invoke a two-way operation to ensure that
//        it is initialized before using it
//
myAllocator.doInitialization();
```

See also

the section called “ create” on page 213 , the section called “ delete” on page 217

declare

Declare a variable.

Syntax

```
declare type identifier { [ integerLiteral ]   assignmentExpression };
declare { ( type identifier { [ integerLiteral ]
                                                assignmentExpression }; )+ }
declare const type identifier { [ integerLiteral ]   assignmentExpression };
```

Description

All variables must be declared before they are used. The KIS action language is strongly typed. Variables are scoped to an operation, state, or code block defined within an operation or state.

The value of a variable is undefined until it is assigned a value. The only exception is object handle which are initialized to empty when they are declared. Variables can be assigned initial values in the declare statement.

Variables can be declared as arrays. There is no support for initializing the values of an array in the declare statement.

Strings can also be declared as either bounded or unbounded.

IDLos constraints

None.

Exceptions

None.

Warnings

A variable should always be initialized to a known value. The value of an uninitialized variable is undefined and may change in the future. The only exception is for object handles, which are empty after being declared.

Example

```
//
// Declare some variables
//
declare long      aLong = 0;
declare AnEntity  anEntity;

//
// Declare an array of shorts
//
declare short     anArrayOfShorts[100];

//
// Declare a variable inside of a for loop
//
declare long i;
for (i = 0; i < 10; i++)
{
    declare long tmpVar;
}
// tmpVar is out of scope here

//
// Bounded strings
//
declare string<10> firstName;

//
// Unbounded string
```

```
//
declare string  unboundedString;
```

See also

the section called “ Types” on page 240

delete

Destroy an object.

Syntax

```
delete handle;
```

Description

Destroy an object. All memory associated with the object is discarded. After invoking delete on an object handle, using that handle will cause a system exception to be thrown.

An implicit unrelate is performed on all relationships in which this object participates.

IDLos constraints

None.

Exceptions

```
swbuiltin::ExceptionResourceUnavailable
```

A resource required to create the object was unavailable. This can happen if this is a delete of a remote or a persistent object.

Warnings

Be careful when deleting one of multiple references to an object: any outstanding references will be invalid once the object is deleted. However, within the same transaction as the delete the object reference remains valid, as shown in the following example. This example will print “Setting value” when the model is run:

```
package P
{
    entity E
    {
        attribute long i;
    };

    [ local ] entity StartUp
    {
        [ initialize ] void doIt();
    };

};

action P::StartUp::doIt
```

```
{`
    declare E e1;
    declare E e2;

    create e1;

    // Make both handles refer to same instance of E
    e2 = e1;

    // Instance will be deleted when transaction commits.
    // Meanwhile the reference is still valid
    delete e1;

    // e2 refers to an object scheduled for deletion,
    // but right now the reference is still valid
    if (!empty e2)
    {
        // This is reached because the reference is still valid
        printf("Setting value\n");
        e2.i = 1;
    }

    // Instance deleted on return (transaction commits)
`};
```

Example

```
declare Customer aCustomer;
create aCustomer;

// Do something with customer

// Destroy the customer
delete aCustomer;

// aCustomer handle is now invalid
```

See also

the section called “ create” on page 213 , the section called “ create singleton” on page 214

empty

Test and set handles to empty.

Syntax

```
declare type_identifier handle = empty;
empty  handle
handle = empty;
```

Description

empty returns a boolean value if used in an expression. True is returned if the handle is currently unassigned to a valid object reference. False is returned if the handle holds a valid object reference.

empty can also be used to initialize a handle to known value that indicates that the handle does not contain a valid object reference.

IDLos constraints

None

Exceptions

None.

Warnings

None

Example

```
entity Customer { };

...

declare Customer aCustomer = empty;

//
// If the customer handle is empty create a new customer
//
if (empty aCustomer)
{     create aCustomer;
}

//
// Set the customer handle to empty
//
aCustomer = empty;
```

See also

Exceptions

Exception handling.

Syntax

```
try statementBlock ( catchStatement )+
catch( scopedType ) statementBlock
throw handle;
```

Description

These statements provide support for handling user and system exceptions.

User exceptions are defined in IDLos using the IDL exception and raises keywords. System exceptions are a predefined set of exceptions that can be raised by the KIS runtime and adapters. System exceptions cannot be thrown by the user, but they can be caught.

The supported system exceptions are defined in the swbuiltin component. You must import this component to catch system exceptions in action language. For details on the individual system exceptions, refer to the on-line documentation for the swbuiltin component.

IDLos constraints

User exceptions are defined using IDLos. Operations that throw an exception must have a raises clause in the operation signature.

Exceptions

None.

Warnings

Uncaught user and system exceptions will cause the current oneway operation to be discarded or the engine will exit with an error depending on a configuration value.

System exceptions should not be thrown by users.

Handling the system exceptions ExceptionDeadLock and ExceptionObjectDestroyed in application code not executed as part of spawn will produce unpredictable but bad results.

Example

```
package Life
{
   entity A
   {
       exception IsDead
       {
    string  causeOfDeath;
       };
       exception IsTired
       {
    long    nextTime;
       };
       void wakeUp(in string reason) raises (IsDead, IsTired);
       void checkIt(void);
    };
;};

//
//  wakeUp operation.  Throws exception on how
//  the wakeup was processed
//
action Life::A::wakeUp
{`
    //
    //    Figure out my current state
    //
    if (imDead)
    {
        declare IsDead          id;
 id.causeOfDeath = "boredom";
        throw id;
    }
    if (imTired)
    {
```

```
        declare IsTired         it;
 it.nextTime = 60;
        throw it;
    }
`};

//
//        checkIt handles exceptions thrown by wakeUp
//
action Life::A::checkIt
{`
    try
    {
        self.wakeUp("it's noon!");
 }
    catch (IsDead id)
    {
        declare string cd = id.causeOfDeath;
 printf("id.causeOfDeath = %s\n", cd.getCString());
    }
    catch (IsTired it)
    {
 printf("it.nextTime = %s\n", it.nextTime);
    }
`};

//
//   Example of string overflow exception
//
entity StringOverflow
{
    attribute string<5>  stringFive;
};

//
//   Assign a string larger than 5 characters to
//   stringFive.  This will cause an exception
//
try
{
    self.stringFive =”123456”;
}
catch (ExceptionStringOverflow)
{
    printf(“caught ExceptionStringOverflow\n”);
}

//
//   Example invalid array exception
//
typedef long ArrayFive[5];

//
//   Use an index > 4.  This will cause an exception.
//   Remember that arrays are zero based.
//
declare ArrayFive arrayFive;
try
{
    arrayFive[5] = 1;
}
catch (ExceptionArrayBounds)
{
    printf(“caught ExceptionArrayBounds\n”);
}
```

See also

the section called “spawn” on page 233 , the section called “ Transactions” on page 238

for

Iterate over statements with iterator expression and termination condition.

Syntax

```
for ( lval { assignmentExpr } ; conditionExpr ; iteratorExpr )
      statementBlock
```

Description

An iteration statement; enables you to loop through and execute a block of code until a condition becomes false.

assignmentExpr is performed once before iteration begins. It is typically used to set the initial value of a variable used in the next two expressions.

conditionExpr is evaluated before each iteration. If the expression is nonzero, then statementBlock is executed.

iteratorExpr is performed each iteration after statementBlock is executed.

IDLos constraints

None.

Exceptions

None.

Warnings

None.

Example

```
for ( x = 0; x < maxTimes; x++ )
{
    //
    //    Repeat this work maxTimes
    //
}
```

See also

the section called “ for ... in” on page 223 , the section called “while” on page 242

for ... in

Iterate over an extent or relationship.

Syntax

```
for handle in setSpec { where whereSpec } statementBlock
```

Description

Iterates over a set of objects. If the set is empty, the statement block is never executed. The handle must be of the same type as the set it iterates over.

IDLos constraints

None.

Exceptions

```
swbuiltin::ExceptionStringOverflow
```

The length of a bounded string attribute was exceeded in a value specified for that attribute in the where clause.

Warnings

None.

Example

Consider the following IDLos definitions:

```
entity Customer { attribute string firstName; };
entity Queues { };

relationship CustomerQueues
{
    role Queue holds 0..* Customer;
    role Customer waitsIn 1..1 Queue;
};
```

Queues and Customers that have been created and related can be iterated across using the for...in statement in various ways:

```
//
// --- welcome all the customers held in some queue q ---
//
for cust in q->Customer[holds]
{
    //    Welcome this customer
}

//
// --- Send a bill to every customer that exists ---
//
```

```
for cust in Customer
{
    //    Send a bill to this customer
}

//
// --- Welcome customers called Dan in some queue q ---
//
for cust in q->Customer[holds] where (cust.firstName == “Dan”)
{
    //    Welcome this Dan
}
```

See also

the section called “cardinality” on page 211 , the section called “select” on page 229 , the section called “in” on page 225

if else else if

Conditionally execute a statement block.

Syntax

```
if booleanExpression
    statementBlock
  { ( else
           statementBlock   elseIfStatement ) }

elseIfStatement :
else if booleanExpression
    statementBlock
  { ( else
      statementBlock   elseIfStatement ) }
```

Description

A conditional control flow operator. The else clause is optional. If statements may be nested.

IDLos constraints

None.

Exceptions

None.

Warnings

None.

Example

```
if (aCustomer.balanceDue < aCustomer.creditLimit)
{
    //
    //    Allow withdrawal
    //
}
else if (aCustomer.isPaymentOverdue)
{
    //
    //    Refuse order with rude message
    //
}
else
{
    //
    //    Credit limit is expired, but payments
    //    current.  Politely refuse withdrawal
    //
}
```

See also

the section called “ for” on page 222 , the section called “while” on page 242

in

Test whether a given object reference refers to a member of some set of objects.

Syntax

```
handle in handleOrSelf  ->  chainSpec
```

Description

Returns true if the object referenced by the handle is a member of the set specified. Otherwise it returns false.

IDLos constraints

None.

Exceptions

None.

Warnings

Do not confuse this function with the for..in statement.

Example

```
//
// If is one of my cars, drive it
//
if( thisCar in self->Car[owns] )
{
    thisCar.drive();
}
```

See also

the section called “cardinality” on page 211 , the section called “ for ... in” on page 223

isnull

Test and set an attribute or struct member’s null indicator.

Syntax

```
isnull attributeName
attributeName = isnull;

isnull structInstance.memberName
structInstance.memberName = isnull;
```

Description

isnull returns a boolean value if used in an expression. true is returned if the attribute or structure member has never been assigned a value. false is returned in the attribute or structure member has been explicitly set. For object attributes, values provided at create time as well as initial values defined in the entity will also clear the null indicator.

isnull can also be used to set the null indicator of an attribute or structure member.

IDLos constraints

None.

Exceptions

None.

Warnings

If isnull is used to set the null indicator of an attribute or structure member the data for the attribute or structure member does not change though subsequent tests for isnull will be true.

Example

```
struct Item
{
```

```
 string name;
}
entity Invoice
{
   ...
    attribute long total;
    attribute Item item;
}
declare Invoice anInvoice;
create anInvoice;
...
declare Item item;
if (isnull anInvoice.total)
{
     // the attribute total is null.
}

item = anInvoice.item;
assert(isnull item.name);
```

relate

Relates one object to another across a relationship.

Syntax

```
relate (handle   self) relationshipSpec (handle   self)
       { using  (handle   self) } ;
```

Description

Establishes a relationship between two objects, or an associative relationship between three objects if a using clause is provided.

Relationships can be used as ordered lists by defining a 1..1 relationship from an entity to itself. The order of the handles in the relate statement defines the order of the objects in the relationship.

The order of the two handles is significant only if they refer to the same entity.

If the cardinality of any side of a role is 1, performing a relate on that role will cause an implicit unrelate to occur for any objects previously in that relationship.

IDLos constraints

None.

Exceptions

None.

Warnings

None.

Example

```
entity Customer { };
entity Queues { };

relationship CustomerQueues
{
    role Queue holds 0..* Customer;
    role Customer waitsIn 1..1 Queue;
};

relationship CustomerOrder
{
    role Customer inFrontOf 1..1 Customer;
    role Customer inBackOf 1..1 Customer;
};

//
//    Put a customer in a queue
//
relate aCustomer waitsIn aQueue;

//
//    This is also valid for putting a customer
//    in a queue
//
relate aQueue holds aCustomer;

//
//    Put impatient customer in front of nice customer
//
relate impatientCustomer inFrontOf niceCustomer;
```

See also

the section called “ unrelate” on page 241

return

Return from an operation or state.

Syntax

```
return { expression } ;
```

Description

A return statement signals completion of an operation or a state and returns control to the caller. Operations with return values must have a return statement and provide a return value expression.

There may be any number of return statements in an operation or state.

IDLos constraints

Operations must be declared with the correct return type.

Exceptions

None.

Warnings

It is illegal to return a value from a state.

Example

```
entity ATM
{
    long getCash();
    void pressOk();
};

//
//    Implementation of getCash operation
//
action getCash
{‘
    declare long        cashAmount = 12345;
    return cashAmount;
‘};

//
//    Implementation of pressOk operation
//
action pressOk
{‘
    return;
‘};
```

See also

select

Select an object using a relationship or an extent.

Syntax

```
select handle from (handle   self) ( ( -> ) chainSpec )+
             { where whereSpec };
select handle from className where whereSpec;
select handle from singleton className;
```

Description

select always returns a single object. It can be used against an extent or a many relationship only with a where clause to narrow the select to return a single object.

The three forms of select above are:

- navigate a relationship chain with a where clause. The where clause is not required if this navigation is towards the one side of a relationship role.

- select a single object from an extent using a where clause.
- select a singleton.

Constraints

None.

Exceptions

```
swbuiltin::ExceptionStringOverflow
```

The length of a bounded string attribute was exceeded in a value specified for that attribute in the where clause.

Warnings

It is a build audit if a select statement returns more than a single object.

Example

```
[ singleton ] entity TheBoss { };
entity Customer
{
    attribute long  custId;
    key CustomerId { custId };
};
entity Contact { };

relationship CustomerContacts
{
    role Contact isFrom 1..1 Customer;
    role Customer has 0..* Contact;
};

///// ---- above was IDLos, below is action language ---- /////

//
//    Select the next customer in line
//
select aCustomer from aContact->Customer[isFrom];

//
//    Select customer 123 from all customers
//
select aCustomer from Customer where (aCustomer.custId == 123);

//
//    Find the boss
//
select boss from singleton TheBoss;
```

See also

```
the section called “  for
    ... in” on page 223
```

select...using

Select an object using a key.

Syntax

```
select handle from className using keyName where whereSpec
          { readlock   writelock   nolock }
          { on empty create { values valuesSpec } }

select handle from singleton className using keyName where whereSpec
          { readlock   writelock   nolock }
          { on empty create { values valuesSpec } }
```

Description

select...using looks up an entity using the specified key. By default, if the object is found, it is locked for reading. If you specify nolock, the object will not be locked. Specifying writelock locks the object for writing.

If you specify on empty create and the object is not found, select...using creates a new object with the key values you supplied in the where clause. You can set other attribute values using the optional values clause.

Warnings

If no object is found, and on empty create was not specified, it is possible that an entity matching the specification may be created in another transaction. This could cause a subsequent create of the object to throw ExceptionObjectNotUnique.

If is a build audit failure if select can return more than one instance.

Exceptions

```
swbuiltin::ExceptionStringOverflow
```

The length of a bounded string attribute was exceeded in a value specified for that attribute in the where clause.

Examples

The following examples are based on this IDLos entity definition:

```
entity Obj
{
    attribute long x;
    attribute long y;

    key key1 { x };
};
```

To select the instance of entity Obj where x is zero:

```
declare Obj obj;
select obj from Obj using key1 where (obj.x == 0);
```

If no instance exists where x is zero, the handle "obj" will be empty. To create an object with the given key if it is not found, use on empty create:

```
select obj from Obj using key1 where (obj.x == 0)
on empty create;
```

This instantiates an object with the attribute x initialized to zero. The attribute y will be unset. To initialize additional attributes when the entity is created, a values clause may be used:

```
select obj from Obj using key1 where (obj.x == 0)
on empty create values (y:46);
```

Note that the key attributes' values are taken from the where clause, and may not be specified again in the values clause.

Locking examples

By default, when an entity is found, it is locked for reading. This ensures that the entity will not be modified by another transaction after it is selected. If the entity is to be modified, a write lock may be taken via the writelock option:

```
select obj from Obj using key1 where (obj.x == 0)
writelock;
```

This prevents a lock promotion if a write lock is needed later, eliminating the possibility of a lock promotion deadlock.

The nolock option may be used to specify that the object should not be locked. This is especially useful when sending oneway events to objects which may be locked by another transaction:

```
select obj from Obj using key1 where (obj.x == 0)
nolock;
```

Using the nolock option means that another transaction can modify the object you have just selected, while you are handling it.

self

Reference the current object.

Syntax

```
self
```

Description

The self keyword is used to reference the current object in which an operation or state is executing.

In spawned operations, self refers to the thread of execution, not to an actual object. See the section called "spawn" on page 233 and "Terminating spawned threads" on page 227 for more information.

IDLos constraints

None.

Exceptions

None.

Warnings

Assigning empty or any handle to self is illegal.

Example

```
entity Customer
{
    attribute long  customerId;
    void anOperation();
};

action anOperation
{‘
    declare long x;
    x = self.customerId; // Access customerId in the current object
‘};
```

See also

the section called “ empty” on page 218

spawn

Spawn a user thread of control.

Syntax

```
spawn scopedType ( { paramSpec ( , paramSpec )* } );
```

Description

Spawn is used to create a thread of control that is managed by the user. This provides a mechanism for users to call blocking external functions and manage transactions as work is injected into KIS.

For example, spawn can be used to create a listener thread for a protocol stack. As messages are received on the protocol stack a transaction is started using the transaction support in action language and the work is injected into KIS.

The spawn statement can only be used on operations in local entities. Thus the spawned operation is not associated with any transaction or shared memory.

Object references used as parameters to the spawn operation and ones declared in the spawn operation are not in a transaction until a begin transaction statement is executed. Access to these objects remains valid until a commit or abort transaction statement is executed. Accessing these objects outside of a transaction will cause a runtime exception.

For example, the following action language will cause a runtime exception:

```
ALocalEntity::aSpawnMethod(in long arg1)
{
    declare AnEntity    anEntity;

    // Causes a runtime exception since no transaction
    //    was started
    create anEntity;

    begin transaction;
        ...
}
```

IDLos constraints

You can only spawn an operation that:

- is defined in a local entity
- is a two-way void operation
- uses only in parameters

Exceptions

None.

Warnings

The spawn verb in action language is non-transactional. This implies that if the spawn verb is called in an operation that is retried because of deadlock, the spawn of the original thread will already have happened and will not be rolled back. Modelers must take caution to ensure that operations that are using spawn do not get retried.

Spawned operations must use begin transaction before accessing any objects. Similarly, they must use commit transaction once they have finished.

Spawned operations must explicitly manage all exception handling, including transaction deadlocks. (A transaction deadlock can occur when accessing an object reference, accessing an attribute, or invoking a two-way operation.) The spawned operation must catch the ExceptionDeadLock system exception and abort the current transaction. Any uncaught system exceptions will cause a fatal engine failure.

Spawned operations cannot use return while in an active transaction. Doing so causes a fatal engine error.

When an engine is shut down, the thread manager kills spawned threads that are not in an active transaction. During transactions in spawned threads, you should periodically check whether the thread needs to terminate. Refer to the online documentation for the swbuiltin component to learn more about using the shouldTerminate() operation.

A spawned thread that blocks in an external function with a transaction active can prevent clean shutdown. Do not block within transactions in spawned threads.

Example

This is a simple example of how the spawn and transaction statements can be used to implement a simple server.

```
package AServer
{
    [ local ]
    entity Initialize
    {
        [ initialize ]
        void startServer(void);
    };

    [ local ]
    entity ServiceThread
    {
        void runServer(in long msgSize);
        void doWork(inout RuntimeServer rs);
    };

    //
    //    The following entity method implementations are
    //    left as an exercise to the reader.
    //
    entity WorkerObject
    {
        oneway void doit(in string data);
        ...
    }

    [ local ]
    entity RuntimeServer
    {
        exception NetworkFailure { };
        void init(in long msgSize) raises (NetworkFailure);
        void recv(out string data) raises (NetworkFailure);
    };
};

//
//    Implementation of startServer
//
action AServer::Initialize::startServer
{`
    //
    // Assumes a component to read the registy
    //
    declareregmgr::Registry   registry;
    declare long       msgSize;
    msgSize = registry.getInt32("SocketServer", "msgSize", 0);
    declare long    numThreads;
    numThreads = registry.getInt32("SocketServer","numThreads", 5);

    //
    // Spawn numThreads threads of control
    //
    declare long    i;
    for (i = 0; i < numThreads; i++)
    {
        spawn ServiceThread::runServer(msgSize:msgSize);
    }
`};
//
```

```
//    Implementation of runServer
//
action AServer::ServiceThread::runServer
    {`
        //
        // Call native method to do server initialization
        //
        declare RuntimeServer   rs;
        declare boolean         runForever = true;
        while (runForever)
        {
            declare boolean     initComplete = false;
            try
            {
                rs.init(msgSize:msgSize);
                initComplete = true;
            }
            catch (RuntimeServer::NetworkFailure)
            {
                printf("Could not init server\n");

                //
                //    Sleep and retry the server
                //    initialization.
                //
                swbuiltin::nanoSleep(60, 0);

                //
                // Or we could kill the engine here.
                //
                // swbuiltin::stop(1);
                //
            }

            //
            // Call dowork to process work until the
            // connection is lost. This shows that we can
            // call  other  IDLos local entity  operations
            // in a spawned thread of control.
            //
            if (initComplete == true)
            {
                self.doWork(rs);
            }
        }
    `};

action AServer::ServiceThread::doWork
    {`
        //
        // Create a worker object. Probably actually want
        // to create a pool of them, and manage a free list
        // (via a relation) for injecting work.
        //
        declare WorkerObject    wo;

        //
        // Need to manage lifecycle of this object.
        //
        begin transaction;
        create wo;
        commit transaction;

        declare boolean         ok = true;
        while (ok)
        {
            try
```

```
            {
                declare string  data;

                //
                // Assume this call reads in data
                // from the network via the ::recv()
                // socket call. Since it blocks, we don't
                // want to be in a transaction.
                //
                rs.recv(data);

                //
                // Inject work into KIS (see
                // note above about worker pool).
                //
                begin transaction;
                wo.doit(data);
                commit transaction;
            }
            catch (swbuiltin::ExceptionDeadLock)
            {
                //
                // Here we throw away work and try
                // another recv(). Could be updated to
                // continually try and post the last work
                // item.
                //
                printf("got deadlock\n");

                //
                // Abort any pending work.
                //
                abort transaction;
            }
            catch (RuntimeServer::NetworkFailure)
            {
                //
                // Example of catching a user defined
                // exception. Note we don't call
                // abort, since this was thrown from the
                // recv call.
                //
                printf("Caught a network failure\n");
                ok = false;
            }
            //
            //    Should catch and abort on all system
            //    exceptions
            //
        }

        //
        //    returning here causes the network
        //    code to restart.
        //
        return;
    `};
```

See also

the section called “ Exceptions” on page 219, the section called “ Transactions” on page 238

test_assert

Model level assertion.

Syntax

```
test_assert ( expression );
```

Description

expression can be any valid boolean expression defined in IDLos.

When an assertion fails, the runtime logs an error message as a fatal error in the trace file and sends the error message to `stderr` . The message includes the model file name and line number where the assertion failed.

The following is an example of an assertion message:

```
x.soc:59: failed assertion '(idx > 1)'
```

After the runtime logs the message, it throws a runtime exception in the model.

IDLos constraints

None.

Exceptions

None.

Warnings

Test assertions are always enabled.

Examples

```
test_assert( !empty myobj );
test_assert( idx >= 1 );
test_assert( myEnum == One    myEnum == Two );
test_assert( self.boolOp() == true );
```

Transactions

Explicitly manage KIS transactions.

Syntax

```
( begin   commit   abort ) transaction ;
```

Description

The three types of transaction statements let you explicitly manage transactions. This is only required and only allowed in an operation that was invoked using the spawn statement. All other operations and states are implicitly in a transaction when they are executed.

Objects that are created or selected in a spawned operation must be done in the context of a valid transaction that was started using begin transaction.

Once an object has been created or selected in a valid transaction it is implicitly reattached to any new transactions

See also “Transaction processing” on page 243 and related topics for a discussion of how transactions, locks, and deadlocks are handled at run time.

IDLos constraints

None.

Exceptions

```
swbuiltin::ExceptionDeadLock
```

A deadlock was detected. The current transaction should be aborted and retried.

```
swbuiltin::ExceptionObjectDestroyed.
```

An object accessed in the current transaction is being destroyed in another transaction. The current transaction should be aborted and retried.

Warnings

Accessing objects passed as parameters are declared in a spawn operation that are not in a valid transaction will cause a runtime exception.

Example

This is a simple example of creating and selecting objects in a spawned local operation. A more detailed example of using transaction control is also provided under the section called “spawn” on page 233.

```
ALocalEntity::aSpawnOperation(in long arg1)
{
    declare MyObject     mobj;
    declare SelectObject sobj;

    //
    //    Create and select in a valid transaction
    //
    begin transaction;
    create mobj;
```

```
    select sobj from singleton SelectObject;
    commit transaction;

    for (;;)
    {
        begin transaction;

        //
        //    No need to select these objects again.
        //    They are implicitly reattached to the new
        //    transaction.
        //
        mobj.numThings++;
        sobj.doIt(mobj.numThings);

        commit transaction;
    }
}
```

See also

the section called " Exceptions" on page 219, the section called "spawn" on page 233

Types

Action language data types.

Syntax

See IDLos chapter.

Description

The action language uses the IDL type system, just like the rest of IDLos does. The declare statement is used to define a type in an operation or state.

Structure members are accessed using the "." notation.

Arrays and sequences are both accessed using "[n]" indexing to access a specific element in an array or sequence. Arrays and sequences use zero based indexing.

Unbounded sequences are automatically grown at runtime as required. If the index being accessed is larger than the current sequence size, the sequence is resized. When a sequence is resized, the values of all new sequence members is undefined. They must be initialized to a known value by the application.

IDLos constraints

None.

Exceptions

```
swbuiltin::ExceptionArrayBounds
```

An array bounds error was detected by the runtime.

```
swbuiltin::ExceptionStringOverflow.
```

A string overflow was detected by the runtime. This can happen when the size of a bounded string is exceeded.

Warnings

None.

Example

```
struct AStruct
{
    long    aLongMember;
    short    aShortMember;
};

typedef sequence<char> UnboundedCharSequence;
typedef long ALongArray[10];

..... Above was IDLos, below is action language .....


// --- Accessing a structure member ---
declare AStruct    aStruct;
declare long       aLong;
aLong = aStruct.aLongMember;

// --- Accessing index 23 in unbounded sequence.  ---
// --- NOTE - Sequences and arrays are zero based. ---
declare UnboundedCharSequence   seq;
declare char                    aChar;
aChar = seq[22];

// --- Accessing the first index in an array ---
declare ALongArray   array;
declare long         aLong;
aLong = array[0];

// --- Using an any ---
declare any    anAny;
declare long   aLong;
declare string aString;

// --- Assign a long to an any ---
aLong = 1;
anAny <<= aLong;
// anAny now contains 1

// --- Now assign a string to the same any ---
aString = “hello world”;
anAny <<= aString;
// anAny now contains “hello world”
```

See also

the section called “declare” on page 215 , the section called “ Exceptions” on page 219

unrelate

Unrelate objects.

Syntax

```
unrelate  (handle   self) relationshipSpec  (handle   self)
          { using  (handle   self) } ;
```

Description

The unrelate statement terminates the relationship between the specified objects. If the relationship is an associative relationship the using clause must be specified to define the third-party in the associative relationship. Deleting a related object performs an implicit unrelate.

IDLos constraints

None.

Exceptions

None.

Warnings

None.

Example

```
entity Customer { };
entity Queues { };

relationship CustomerQueues
{
    role Queue holds 0..* Customer;
    role Customer waitsIn 1..1 Queue;
};

..... Above was IDLos, below is action language .....

// --- Remove a customer from a queue ---
unrelate aCustomer waitsIn aQueue;

// --- Remove a customer from a queue the other way works too ---
unrelate aQueue holds aCustomer;
```

See also

the section called “ delete” on page 217 , the section called “relate” on page 227

while

Loop over a statement block while a condition remains true.

Syntax

```
while booleanExpression statementBlock
```

Description

The while statement loops and executes a set of statements repeatedly as long as a condition is true. You may use break to exit a while loop at any time, or use continue to skip immediately to the next iteration.

IDLos constraints

None.

Exceptions

None.

Warnings

None.

Example

```
while (aClerk.atWork)
{
    // --- Can this clerk help this customer? ---
    if (!aCustomer.canHelp)
    {
        continue;
    }

    // --- Time to quit?  Just leave ---
    if (aClerk.doneForTheDay)
    {
        break;
    }

    //
    //    Service customers
    //
}
```

See also

the section called “ break” on page 210 , the section called “ continue” on page 212 , the section called “ for” on page 222

Complete action language grammar

This section contains the grammar that is used by the action language parser. Grammatical elements in this list occasionally differ from their equivalents in the action language statement and functions section, because in that section a few changes were made for the sake of clarity:

- minor name changes added missing semantic information
- some elements were expanded or condensed according to the grammar

The elements defined in the grammar are listed in alphabetical order. A few elements—such as string-literal or identifier—are defined by the IDL grammar. See the IDLos grammar in Chapter 2 for a definition of these items.

```
action :
 type scopedMethodName inputParameterList { const } statementBlock
 includeStatement   idStatement   package quotedIdentifier ;

actionFile :
 ( action )+

addingExpression :
 multiplyingExpression ( ( +   - ) multiplyingExpression )*

arrayExpression :
 { :: } ( quotedIdentifier :: )* quotedIdentifier   integerLiteral

arrayList :
 ( [ expression ] )*

assertStatement :
 ( assert booleanExpression | test_assert booleanExpression )

assignEqualOp :
 *=   /=   %=   +=   -=   <<=   >>=   &=
 ^=    =

assignmentExpression :
 ( =   assignEqualOp   shiftOp ) expression   ( ++   -- )

attributeName :
 quotedIdentifier

binaryOperator :
 +   -   *   /

bitExpression :
 signExpression ( ( &       ^ ) signExpression )*

booleanExpression :
 expression

booleanLiteral :
 TRUE   true   FALSE   false
booleanOperator :
 &&

cType :
 unsigned short   short   unsigned int   int
 long   long long   unsigned ( long
 long long )   float   double   unsigned char
 char   void

cardinalityFunction :
 cardinality ( scopedType   handleOrSelf -> chainSpec )

castExpression :
 castOperator < { const } ( scopedType   cType ) { * } >
 expression

castOperator :
```

```
 const_cast   dynamic_cast   reinterpret_cast
 static_cast

catchExpression :
 scopedType { quotedIdentifier }

catchStatement :
 catch catchExpression statementBlock

chainSpec :
 scopedType [ relationshipSpec ]

charLiteral :
 character-literal

className :
 scopedType

comparisonOperator :
 <   >   <=   >=   ==   !=

constDeclare :
 { const }

createStatement :
 create { singleton } handle { on locationSpec }
  { implementation implementationSpec }
  { values valueSpec ( , valueSpec )* }
declareSpecStatement :
 type quotedIdentifier { < integerLiteral > }
 {
  [ arrayExpression ] ;
  assignmentExpression ;
  ;
 }

declareStatement :
 declare declareSpecStatement
 declare { ( declareSpecStatement )+ }

deleteStatement :
 delete handle
 delete [ ] handle

elseIfStatement :
 else if booleanExpression statementBlock
 { ( elseStatement   elseIfStatement ) }

elseStatement :
 else statementBlock

emptyFunction :
 empty { handleOrSelf { . attributeName } { { expression } } }

expression :
 relationalExpression ( booleanOperator relationalExpression )*

extendedChain :
 ( ( -> ) chainSpec )+

externStatement :
 extern ( scopedType   cType ) quotedIdentifier

floatLiteral :
 floating-point-literal

forStatement :
```

```
 for handle in setSpec
  { where whereSpec }
  { order by attributeName } statementBlock
 for lval { assignmentExpression } ; expression ; lval { assignmentExpression }
   statementBlock

genAssignDeclareStatement :
 { const } ( cType   scopedType   self )
  ( paramList
    userDereference quotedIdentifier
      { ( [ arrayExpression ]   assignmentExpression   paramList ) }
    . quotedIdentifier arrayList ( paramList
                                ( . quotedIdentifier )* = expression
                                assignEqualOp expression ) )

handle :
 quotedIdentifier

handleOrSelf :
 handle   self

idStatement :
 quotedIdentifier expression ;

ifStatement :
 if booleanExpression statementBlock { ( elseStatement   elseIfStatement ) }

implementationSpec :
 identifier   integerLiteral

inFunction :
 handle in handleOrSelf -> chainSpec

includeStatement :
 include filename

inputParameter :
 type & quotedIdentifier

inputParameterList :
 ( inputParameter ( , inputParameter )*   void   )

integerLiteral :
 decimal-integer-literal   hexadecimal-integer-literal   octal-integer-literal
 decimal-integer-literal_LL   hexadecimal-integer-literal_LL
 octal-integer-literal_LL

isnullFunction :
 isnull { handleOrSelf . attributeName }


literal :
 stringLiteral   charLiteral   integerLiteral   floatLiteral   booleanLiteral

locationSpec :
 identifier   integerLiteral

loopControlStatement :
 break   continue

lval :
 { ( ++   -- ) } quotedIdentifier arrayList

lvalStatement :
 lval { assignmentExpression }

methodName :
```

```
 quotedIdentifier

methodOrVar :
 handleOrSelf . attributeName (. attributeName )* ( paramList   arrayList )
 handle -> methodName { paramList }
 { ( &   * ) } { :: } ( quotedIdentifier :: )* identifier
     ( paramList   arrayList { ( ++   -- ) } )

multiplyingExpression :
 bitExpression ( ( *   /   % ) bitExpression )*

nameValue :
 attributeName : expression

nativeType :
 unsigned short   short   unsigned ( long   long long )
 long   long long   float   double   boolean   char
 wchar   octet   Object   any   string { < integerLiteral > }
 wstring { < integerLiteral > }   void

paramList :
 { paramSpec ( , paramSpec )* }

paramSpec :
 parameterName : expression   expression

parameterName :
 quotedIdentifier

primitiveExpression :
 literal   emptyFunction   isnullFunction   swalFunction   castExpression
 methodOrVar   expression
 sizeof ( quotedIdentifier   cType )
 new ( { :: } ( quotedIdentifier :: )* quotedIdentifier
 cType ) { ( [ arrayExpression ]   paramList ) }
 self

quotedIdentifier :
 identifier   string-literal

relateStatement :
 relate handleOrSelf relationshipSpec handleOrSelf { using handleOrSelf }

relationalExpression :
 shiftExpression ( comparisonOperator shiftExpression )*

relationshipName :
 quotedIdentifier

relationshipSpec :
 relationshipName

returnStatement :
 return { expression }

scopedMethodName :
 ( quotedIdentifier :: )+ quotedIdentifier

scopedType :
 { :: } ( quotedIdentifier :: )* quotedIdentifier

selectStatement :
 select { distinct className } handle from
  ( handleOrSelf extendedChain { where whereSpec }
     singleton className
     className ( where whereSpec   usingClause )
   )
```

```
setSpec :
 className   handleOrSelf extendedChain

shiftExpression :
 addingExpression ( ( <<   >> ) addingExpression )*

shiftOp :
 <<   >>

signExpression :
 { + - } unaryExpression

spawnStatement :
 spawn scopedType { paramSpec ( , paramSpec )* }

statement :
 declareStatement   createStatement ;   deleteStatement ;
 relateStatement ;   unrelateStatement ;   selectStatement ;   forStatement
 ifStatement   whileStatement   tryStatement   throwStatement ;
 loopControlStatement ;   transactionStatement ;   assertStatement ;
 spawnStatement ;   genAssignDeclareStatement ;   userMethodStatement ;
 lvalStatement ;   returnStatement ;   externStatement ;   statementBlock

statementBlock :
 { ( statement )* }

stringLiteral :
 ( string-literal )+

swalFunction :
 cardinalityFunction   inFunction

throwStatement :
 throw handle

transactionStatement :
 ( begin   commit   abort ) transaction

tryStatement :
 try statementBlock ( catchStatement )+

type :
 constDeclare ( nativeType   userDefinedType )



unaryExpression :
 { unaryOperator } primitiveExpression

unaryOperator :
 !   -   +   ~

unrelateStatement :
 unrelate handleOrSelf relationshipSpec handleOrSelf { using handleOrSelf }

userDefinedType :
 scopedType

userDereference :
 { ( ( * )+   & ) }

userMethodStatement :
 handle ( -> methodName )+ { paramList } { assignmentExpression }

usingClause :
 using className where whereSpec
```

```
   { readlock   writelock   nolock }
  { on empty create { values nameValue ( , nameValue )* } }

valueSpec :
 attributeName : expression   chainSpec : expression

whereExpression :
 ( whereSpec   wherePrimitive )

wherePrimitive :
 ( handleOrSelf . attributeName arrayList   handleOrSelf )
    comparisonOperator primitiveExpression

whereSpec :
 whereExpression ( booleanOperator whereExpression )*

whileStatement :
 while booleanExpression statementBlock
```

11

Build specification reference

This chapter provides a reference for the keywords and commands you use to compose and execute a Kabira Infrastructure Server (KIS) build specification. These appear in alphabetical order. Each page includes the syntax of the keyword or command, a discussion of usage, and an example.

This chapter contains the following sections:

A simple example

The example below shows a very simple build specification. For a full build specification that uses all the standard features, see the section called “Complete build specification example” on page 260.

```
component Component1
{
  source MyIDLosFile.soc
  import ImportedComponent
        {
   importPath=/this/is/where/the/component/is;
```

```
        };
  library=someLibraryThisComponentNeeds;

        package Package1;
        package ::Package2
        adapter sybase
        {
   entity Package1::PersistentEntity
        };
};
```

Understanding the syntax notation

The syntax description for each keyword or command uses the following conventions:

bold screen font indicates a literal item that appears exactly as shown

italics indicates a language element that is defined elsewhere

`plain text` indicates a feature of the notation, as follows:

(item) parentheses group items together

(item)+ the item in parentheses appears one or more times

{ item } items in braces are optional

adapter

Defines an adapter block.

Syntax

```
adapter adapterName { buildLanguage };
```

Description

adapterName is the name of a valid, installed service adapter.

An adapter block defines a service adapter for a specific model element. Each adapter must contain an interface or entity, depending on its type.

Adapters also have properties that affect they way they operate. Refer to the documentation for a specific service adapter for more information on the properties used by that adapter.

Example

```
adapter corba
{
  interface ::MyPackage::anInterfaceA;
  interface ::MyPackage::anInterfaceB;
};
```

component

Defines a component block.

Syntax

```
component identifier  {implementation=service-name}
   { buildLanguage };
```

Description

A component block defines a component being built and will contain all the information needed to build the component. A component may implement one or more packages, or may simply contain interfaces to an external implementation. All packages in a given component must be listed in the order of dependency.

A component can act as a wrapper around a service implemented in another technology. The generated wrapper will behave just like any other component except that it uses a foreign implementation.

serviceName is the name of the foreign service being implemented.

Refer to the documentation for the individual adapter generator for more information on wrapping a foreign service.

Warnings

None

Example

```
component MyComponent
{
    source MyIDLos.soc;
    package MyPackage;
};
component AnExternalComponent implementation=corba
{
  idlPath=MyIDL.idl;
};
```

group

Defines a logical collection of a given type.

Syntax

```
group identifier { buildLanguage };
```

Description

Allows for grouping model references and properties.

Warnings

None

Example

```
component MyComponent
{
    adapter snmp
    {
         group OID
        {
            snmpPackageOID = 1.3.6.1.4.1.3004.2.1.4;
            interface MyPackage::anInterfaceA;
            interface MyPackage::anInterfaceB;
        };
    };
};
```

import

Specifies a component dependency, or aggregates components.

Syntax

```
import identifier;
```

Description

A component may depend on the services provided by another component. You import a component so that its interface definitions and link libraries can be loaded into the Design Center server at build time. (For managing the Design Center Server, see Chapter 14.)

The Design Center always knows where to locate built-in KIS components. If you want to import your own components from an arbitrary location, you can specify this location using the `importPath` property. Note that this property follows normal scoping rules, so you can use different import paths for different components.

Example

```
component MyComponent
{
    // set the include path to the component location
    importPath = /some/include/path;
    import DependentComponentA
};
```

Macros

Allows you to substitute parameters from the command line.

Syntax

Use in the build spec: `$( macro-name )`

Definition on the command line: `swbuild -o macro-name = value`

Description

Before the build specification is parsed, macros are evaluated and substituted with values from the command line. Forms that do not correspond to the syntax are not macros, and no substitution takes place. To explicitly escape the `$(` sequence, you can use `$$(` . Thus `$$(NOTAMACRO)` becomes `$(NOTAMACRO)` after substitution. Because macros are evaluated before parsing, they can be used anywhere:

```
component $(COMPONENT_NAME)
{
     name=$(VALUE);
    $(NAME)=$(VALUE);
    name="Macros can even be in $(STRINGS)";
};
```

An undefined macro evaluates to the name of the macro. For example, if `$(VALUE)` is not defined, it evaluates to `VALUE` , and the Design Center issues a warning.

Example

A typical use of macros is to pass property values from the command line.

```
component dcapiplugin
{
         source=dcapi.soc;
         package dcapi;

         includePath=../include;
         buildType=$(BUILD);
         numParallelCompiles=$(MAXPROCESS);
};
```

The command line associated with the example above would be something like:

```
swbuild -o BUILD=PRODUCTION -o MAXPROCESS=4
```

Warnings

Macros are expanded anywhere, even in comments. This means you can't simply comment out lines containing undefined macros—you must remove the offending macro call.

See also

the section called "swbuild" on page 258

Properties

Specify options in a build specification.

Syntax

```
propertyName = value;
```

Description

Each property is a name-value pair that sets compiler build options for the containing component or adapter.

propertyName is the name of a defined build property from the properties table below.

value is a value as defined in the description column of the properties table below.

The following table is a complete list of project and component properties. For adapter properties refer to the appropriate section in the documentation for the relevant adapter.

Property	Type	Default	Description	Applies to
importPath	directory	.	Search paths used for imported components	project, component, component (import, group)
buildPath	directory	.	Directory for generated output.	project, component
buildType	choice	DEVELOPMENT	One of DEVELOPMENT or PRODUCTION build	project, component
cFlags	string		C compiler flags	project, component
ccFlags	string		C++ compiler flags	project, component
debug	choice	FALSE	Enable action language debug code generation.	project, component
timestamp	choice	FALSE	Enables time stamp collection on an action language, line by line basis	project, component
includePath	directory	.	Search paths used for included files	project, component
ldFlags	string		Linker flags	project, component

Property	Type	Default	Description	Applies to
libraryPath	directory		Search path for libraries	project, component
library	string		Libraries linked with runtime libraries	project, component
methodsPerFile	numeric	50	Number of methods generated per file. Min: 10, Max: 100	project, component
name	string		Name of the generated engine	component
numParallelCompiles	numeric	1	Number of parallel compilations, min. 1	project, component
description	string		Description string for this engine	component
engineGroup	string	Applications	Engine group for this component	component
engineService	string		Add an engine service to this engine.	component
extraArchiveFile	file		Extra file to place in engine archive	component
undef	string		Inserts user-defined undef directives to avoid name conflicts with system headers	component

When you use an adapter factory, additional properties specific to that adapter may also be available. See the relevant adapter factory documentation for these properties and the values they take.

Example

```
// a global property will apply to both components
buildPath = some/build/path;
component MyComponentA
{
        // property local to MyComponentA
        includePath = some/include/path;
};
component MyComponentB
{
    // properties local to MyComponentB
    source = MyIDLosB.soc
```

```
    name=myWonderfulComponent
};
```

source

Specifies a source file for a component.

Syntax

```
source identifier;
```

Description

identifier specifies the filename.

Source files can be added to a component specification at a global or component level. A source file can be an IDLos, action language or header file.

Source files must be listed in the correct order of dependency. It is best to list all source files, including `.act` files, in the specification instead of as a `#include` in the IDLos file. This will optimize Design Center server reloading.

Properties may be associated with a source file, as follows:

- includePath
- libraryPath

Warnings

None.

Example

```
component MyComponent
{
    source /some/source/path/MyIDLos.soc;
    source /some/source/path/MyAL.act;
};
```

swbuild

Build a component from the command line.

Syntax

```
swbuild {-a} {-o macro-name=value} {specification-file}
```

Description

The -a option suppresses building; the Design Center will perform an audit only.

The -o option allows you to set options for macros defined in the build specification.

The specification-file indicates the source file for the build specification. If you omit this argument, the Design Center reads the build specification from standard input.

Example

```
swbuild mybuildspec.osc
```

See also

the section called “Macros” on page 254

Complete build grammar

```
adapterBlock  :
 {  (  propertyStatement     groupStatement     modelrefStatement  )*  }

adapterStatement  :
 adapter  identifier  {  adapterBlock  }  ;

componentBlock  :
 {  (  sourceStatement     importStatement     propertyStatement
     adapterStatement     modelrefStatement     includeStatement  )*  }

componentSpecification  :
 (  componentStatement     propertyStatement  )*

componentStatement  :
 component  identifier  (  implementation  identifier  )*
  {  componentBlock  }  ;

compoundQuotedLiteral  :
 quotedLiteral  (  quotedLiteral  )*

groupBlock  :
 {  (  propertyStatement     modelrefStatement  )*  }

groupStatement  :
 group  identifier  {  groupBlock  }  ;

identifier  :
 [a-zA-Z][a-zA-Z0-9_]*

importStatement  :
 import  identifier  {  propertyBlock  }  ;

includeStatement  :
 include  (  <     "  )  filename (  "     >  )

modelrefStatement  :
 (  package     module     interface     entity  relationship
        operation     signal     attribute     role     key     entity  )
        scoped_name  {  propertyBlock  }  ;

nameToken  :
 identifier

propertyBlock  :
 {  (  propertyStatement  )*  }
```

```
propertyStatement  :
 nameToken  =  {  valueToken  }  ;

scoped_name  :
 {  ::  }  identifier  (  ::  identifier  )*

sourceStatement  :
 source  valueToken  {  propertyBlock  }  ;

unquotedLiteral  :
 literal-content

valueToken  :
 compoundQuotedLiteral     unquotedLiteral
```

Complete build specification example

```
/*
 * Global properties.  These appear outside of any component
 * block, and apply to all components.
 *
 * The following tests global properties and tries to trick
 * our parser with macros and comments.
 *
 */

globalName1=globalValue1;
global$(NAME)2=global$(VALUE)2;
$(GLOBALNAME3)=$(GLOBALVALUE3);

/*comment*/global$(NAME)4/*$(comment)*/=global$(VALUE)4;

// Empty values are allowed. All three of the following are
// empty values:
emptyValue="";
emptyValue=;
emptyValue=$(EMPTYVALUE);

// A component statement describes a component to be built.
// In the plugin service project tree, this maps to ElemEngine

component Component1
{

    localName1=localValue1;
    local$(NAME)2=local$(VALUE)2;
    $(LOCALNAME3)=$(LOCALVALUE3);

    quotedValue = "

    What you see is what you get... almost.

    Escape sequences in this string follow the same
    rules as IDLos.

    So, you can \"escape a string\"
    insert a tab \t, etc.

    Check out unquoted literals if you are specifying file paths.

    ";

    escapedString="Escaped quote:\"  Newline:\n
                                     Escaped backslash:\\";
```

```
quotedCompoundValue =
    "Notice how strings are allowed to "
    "continue, just like in C++;";

quotedMacroValue = "quoted$(VALUE)1";

// Note how unquoted literals don't have any escape sequences.
// This makes file names on NT easy.

pathValue1=/some/file;
pathValue2=C:\some\file;

escapedMacro1 = $$(NOTAMACRO);
escapedMacro2 = "$$(NOTAMACRO)";
escapedMacro3 = "This is $$(NOTAMACRO), OK?";

// The source statement indicates a source file to be
// loaded into the DC server.  Normally source files
// do not need properties.  However, if you are using
// #include in your source file, you may wish to define
// an includePath which only applies to your file.  Be
// aware that using #include is a bad idea.  You should
// list all your files here (both IDLos and Action Language)
// because it allows the DC to optmize reloads.

source MyIDLosFile.soc
{
    includePath = /some/path/name;
};
source MyActionLanguageFile.act;

// The import statement establishes a dependency to
// another component.  The DCAPI plugin will search for
// this component using includePath.  You can specify
// additional includePath, library, etc, parameters
// for this component only by using a property block.

import ImportedComponent
{
    includePath=/this/is/where/the/component/is;
    library=someLibraryThisComponentNeeds;
};

// You must list the packages that will be
// implemented in your componenent.  If leading
// "::" is left off, it is implicit.  You can also
// assign properties to model references.

package Package1;
package ::Package2
{
    someproperty=somevalue;
};

// You may use the services of an adapter to
// bind parts of your model to a particular
// implementation.  It is up to each adapter as
// to what sort of model references and properties
// they allow

adapter sybase
{
   entity Package1::PersistentEntity
   {
       readString="Some SQL string";
       writeString="Some SQL string";
   };
```

```
    };

    // You can have more than one adapter, of course.

    adapter corba
    {
       interface /*comment*/ MySecondPackage::MyPublicInterface;
    };


    // SNMP used groups to specify hierarchical properties
    adapter snmp
    {
        group OID
        {
           snmpPackageOID=1.2.3.4.5.6;
           interface Package1::Interface1;
           interface Package2::Interface2;
           module Package1::Module1;
        };
    };
};


// You can build multiple components in the same
// spec.  Componets can also have different implementations

component Component2 implementation java
{

        // blah, blah

};


// It should be possible to completely parameterize
// a build

component $(Component3)
{
    package $(Package3);
};
```

12

PHP reference

This chapter contains reference pages for each Kabira Infrastructure Server (KIS) PHP extension function call, in alphabetical order. Each PHP extension function call appears in this section on its own page with a top-level syntax definition, a description of the function, and an example.

This chapter contains the following sections:

About the notation

The syntax description for each statement or function uses the following conventions:

bold screen font indicates a literal item that appears exactly as shown

italics indicates a language element that is defined elsewhere

`plain text` indicates a feature of the notation, as follows:

(item) parentheses group items together

(item)* the item in parentheses appears zero or more times

(item)+ the item in parentheses appears one or more times

{ item } items in braces are optional

(item1 item2) the vertical bar indicates either item1 or item2 appears

For example:

```
keyword { optionalKeyword }
    { optionalThing }( zeroOrMoreRepeatingThing )*
    ( oneAlternative   otherAlternative   thirdAlternative )
```

os_connect

Creates a new connection to KIS osw engine.

Syntax

```
$conn_id =
os_connect ($host, $port, $username, $password, $mechanism, $mechanismdata);
$conn_id =
os_connect ($host, $port);
$conn_id =
os_connect ($host);
$conn_id =
os_connect ( );
```

Description

This function creates a connection between the PHP process and the osw engine.

conn_id is a PHP variable.

host and port are optional parameters that locate the Web Adapter engine.

If host and port are not set they will default to the host and port specified in the `php.ini` file.

username, password, mechanism, and mechanismdata are optional authentication parameters. They must be specified if the Web Adapter has secure access enabled.

Warnings

None

Error Conditions

- Not able to connect to KIS server

os_create

Creates a KIS object and returns its reference.

Syntax

```
$objr =
os_create ($conn_id, $scopedName, $attrList);
$objr =
os_create ($conn_id, $scopedName);
```

Description

This function creates an object and returns its reference in the variable objr. If the object has attributes, an optional attrList could be passed in to set the attribute initial values.

When the scopedName type is a singleton, this call acts like the create singleton in action language. It creates the singleton if it does not exist. Otherwise, it returns the object reference.

conn_id is the value returned by `os_connect()` .

objr is the return value.

scopedName is a fully scoped KIS interface name.

attrList is an optional associative array of attribute names and values.

Example

```
//
// create a sales person with “name” initialized
//
$salesType = “myPackage::Salesman”;
$salesAttrArray = array(‘name’ => ‘Slick Willy’);
$salesHandle = os_create($conn_id, $salesType, $salesAttrArray);

//
// create a customer - without attribute array
//
$custType = “myPackage::Customer”;
$custHandle = os_create($conn_id, $custType);
```

Error Conditions

- Not able to connect to KIS server
- Invalid scoped name
- Invalid attribute name
- Unsupported type
- No create access
- Duplicate Key

os_delete

Deletes a KIS object.

Syntax

```
os_delete($conn_id, $objrList);

os_delete($conn_id, $objr);
```

Description

The second parameter may be a either a single object reference or an array containing a list of object references.

conn_id is the value returned by `os_connect()` .

objrList is an array of valid object instances to delete.

objr is a valid object instances to delete.

Warnings

None.

Example

```
//
// Delete all Orders
//
$sn = “mypackage::Order”;
$orderList = os_extent($conn_id, $sn);
os_delete($conn_id,$orderList);

//
// delete a single object
//
$objr = os_create($conn_id, $sn);
os_delete($conn_id, $objr);
```

Error Conditions

- Not able to connect to KIS server
- Invalid handle
- No delete access

os_disconnect

Disconnect from the KIS engine.

Syntax

```
os_disconnect ($conn_id);
```

Description

conn_id is the return value from a call to `os_connect()` .

Warnings

None.

Example

```
/* close a connection to the osw engine */
os_disconnect($xconn);
```

os_extent

Retrieve the extent of object handles of a given type.

Syntax

```
$objrList =
os_extent($conn_id, $scopedName, $attrList);
$objrList =
os_extent($conn_id, $scopedName);
```

Description

This function will select objects of a given type. If attrList is provided, it contains a list of name-value pairs that are ANDed together as a where clause to filter the number of object handles being returned.

If the objects are keyed and attrList has key coverage, a keyed lookup will be performed.

conn_id is the value returned by `os_connect()` .

objrList is the return value and is a PHP array of scalar values.

scopedName is a string containing the fully scoped name of a type.

attrList is an optional associative array of attribute names and values.

Warnings

None.

Example

```
//
// get all customers
//
$sn = “myPackage::Customer”;
$custList = os_extent($conn_id, $sn);

//
// return all customers in California
//
$sn = “myPackage::Customer”;
$whereClause = array(‘state’ => ‘CA’);
$custList = os_extent($conn_id, $sn, $whereClause);
```

Error Conditions

- Not able to connect to KIS server
- Invalid scoped name
- Invalid attribute name
- Unsupported type or bad value
- No extent access

os_get_attr

Retrieves the values of the attributes in a KIS object.

Syntax

```
$attrList =
os_get_attr($conn_id, $objr, $filter);
$attrList =
os_get_attr($conn_id, $objr);
```

Description

This function retrieves attribute values from an object. If filter exists, only the values of those attributes named in the filter array will be returned. Otherwise, all attributes values will be returned.

conn_id is the value returned by `os_connect()` .

attrList is an optional associative array of attribute names and values.

objr is a KIS object handle.

filter is an optional associative array of attribute names and values.

Warnings

None.

Example

```
//
// get all attributes of a cusotmer
//
$attrs = os_get_attr($conn_id, $custHandle);
for  ( reset($attrs);  $name = key($attrs); next($attrs ) )
{
     $avalue = $attrs[$name];
     print “$name = $value\n”;
}

//
// get the state attribute only
//
$filter = array ( “state” => ““ );
$attrs = os_get_attr($conn_id, $custHandle, $filter);
$state = $attrs[“state”];
print “This customer is in state of $state\n”;
```

Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid attribute name

os_invoke

Invoke an operation on a KIS object.

Syntax

```
$returnValue =
os_invoke($conn_id, $objr, $opName, $param, $ex);
$returnValue =
os_invoke($conn_id, $objr, $opName, $param);
$returnValue =
os_invoke($conn_id, $objr, $opName);
```

Description

This function is used to invoke an operation on an object. The returnValue is not required on void operations. If a parameter is an inout or out parameter the value of that array element in param will be modified with the result parameter value after a call. If the operation raises user defined exception, $ex will contain the name of the exception type as a string upon return.

conn_id is the value returned by `os_connect()` .

returnValue is the return value of the operation upon completion.

objr is a valid object instance handle.

opName is the name of an operation.

param is a nested associative array of parameter name and value pairs.

ex is the name of user exception thrown by the operation.

Example

```
//
// call a void operation that does not have any params
//
os_invoke($conn_id, $objr, “runtest”);

//
// call an operation with parameters that returns a boolean
//
$params = array (“name” => “Smith”, “number” => 5);
$ret = os_invoke($conn_id, $objr, “register”, $params);
if ($ret)
{
 print “OK\n”;
}
else
{
 print “register failed\n”;
}
```

When an operation raises user defined exceptions, os_invoke() can get the exception type, but not the exception data if it has member fields.

```
//
//  operation with user defined exceptions
//
$userex = ““;
$params = array();
os_invoke($conn_id, $objr, “myOp”, $params, $userex);
if ($userex != ““)
{
 print “Caught user exception $userex\n”;
}
```

Array and sequence types can be used as in, out, inout parameters and return values.

```
//
//  array or sequence type as out param
//
$params = array ( “myList” => array() );
$ret = os_invoke($conn_id, $objr, “getList”, $params);
$list = $params[“myList”];
for ($i = 0; $i < count($list); $i++)
{
 $value = $list[$i];
 print “list[ $i ] = $value\n”;
}
```

Error Conditions

- Not able to connect to KIS server

- Invalid handle
- Invalid operation name
- Invalid parameter name
- Unsupported type or bad value
- Application exception

os_relate

Relates two interfaces.

Syntax

```
os_relate($conn_id, $fromObjr, $roleName, $toObjr);
```

Description

This function relates two objects using the relationship role roleName.

conn_id is the value returned by `os_connect()` .

fromObjr is a valid object reference handle.

roleName is the name of a role in a relationship between the “from” and “to” objects.

toObjr is a valid object reference handle.

Warnings

None.

Example

```
$salesType = “myPackage::Salesman”;
$salesAttrArray = array(‘name’ => ‘Slick Willy’);
$salesHandle = os_create($conn_id,$salesType, $salesAttrArray);

$custType = “myPackage::Customer”;
$custAttrArray = array( ‘name’ => ‘Lucent’, ‘state’ => ‘NJ’);
$custHandle = os_create($conn_id,$custType, $custAttrArray);
os_relate($conn_id,$salesHandle,  “hasCust”, $custHandle);
```

Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid role name

os_role

Retrieves an array of handles by navigating across a relationship.

Syntax

```
$objrList =
os_role($conn_id, $objr,  $roleNam, $attrList);
$objrList =
os_role($conn_id, $objr,  $roleNam);
```

Description

This function returns an array of object references as it navigates across the appropriate relationship. The name-value pairs in attrList are ANDed together to act as a where clause to filter the number of object handles being returned.

conn_id is the value returned by `os_connect()` .

objrList is a list of object references.

objr is a KIS object handle.

roleName is a fully scoped relationship role name.

attrList is an optional associative array of attribute names and values.

Warnings

None.

Example

```
/*
** assume Salesperson is related 1:M with customer
** and that $salesHandle is already populated with a
** valid Salesperson.
*/
$cList = os_role($conn_id, $salesHandle, “packageName::relationshipName::sellsTo”);
for ( reset($cList);  $cust = current($cList); next($ctList))
{
     /* do something with $cust */
}
```

Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid role name
- Invalid attribute name

- Unsupported type or bad value

os_set_attr

Sets a value in an attribute of a KIS object.

Syntax

```
os_set_attr($conn_id, $objr, $attrList);
os_set_attr($conn_id, $objr, $name, $value);
```

Description

This function sets a value of an object attribute. More than one attribute value can be set in an object.

conn_id is the value returned by `os_connect()` .

objr is a KIS object handle.

attrList is an optional associative array of attribute names and values.

Example

```
//
// set the name of a customer
//
$attrArray = array('name' => 'John');
os_set_attr($conn_id, $objr, $attrArray);

//
// or using
//
os_set_attr($conn_id, $objr, "name", "John");

//
// set array attribute
//
$value = array (1, 2, 3, 4, 5);
$attrArray = array ("longList" => $value);
os_set_attr($conn_id, $objr, $attrArray);
```

Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid attribute name
- Attribute is readonly
- Unsupported type or bad value
- Duplicate Key

os_unrelate

Unrelates two objects.

Syntax

```
os_unrelate($conn_id, $fromObjr, $roleName, $toObjr);
```

Description

This function unrelates two objects that are currently related using the relationship role roleName.

conn_id is the value returned by `os_connect()` .

fromObjr is a valid object reference handle.

roleName is a relationship role name.

toObjr is a valid object reference handle.

Warnings

None.

Example

```
$custList = os_role($conn_id, $salesHandle,  ":sellsTo");
for (reset($custList); $cust = current($custList); next($custList))
{
     os_unrelate($conn_id, $salesHandle,  'hasCust', $cust);
}
```

Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid role name

13

String operations

This reference chapter describes the Kabira Infrastructure Server (KIS) built-in string operations. The operations are presented in the following groups:

Copy and append operations

The operations in this group allow you to copy and append values to a string.

stringCopy()

Copies the string argument to a string.

Syntax

```
void stringCopy(in string src)
```

Exceptions

```
None
```

stringAppend()

Construct a string type from the argument, if not already a string, and append it to the string. There is a version for each supported type:

Syntax

```
void stringAppend(in string src)
void stringAppend(in char[] src)
void stringAppend(in short value)
void stringAppend(in long value)
void stringAppend(in unsigned short value)
void stringAppend(in unsigned long value)
void stringAppend(in boolean value)
void stringAppend(in long long value)
void stringAppend(in unsigned long long value)
void stringAppend(in float value)
void stringAppend(in double value)
void stringAppend(in char value)
void stringAppend(in octet value)
```

Exceptions

```
None
```

formatAppend()

Appends the value to the string. These operations allow you specify a format (width and precision) for the value being appended to the string.

The supported types are:

- double
- long
- string
- unsigned long

Syntax

```
void formatAppend(
   in long width,
   in long precision,
   double value)
void formatAppend(
   in long width,
   in long precision,
   in unsigned long value)
void formatAppend(
   in long width,
   in long precision,
   in long value)
void formatAppend(
   in long width,
   in long precision,
   in string value)
```

Exceptions

```
None
```

formatHexAppend()

Appends a hexadecimal value to a string. This operation allows you to specify a format (width and precision) for the value being appending.

Syntax

```
void formatHexAppend(
   in long width,
   in long precision,
   in unsigned long value)
```

Exceptions

```
None
```

Comparison operations

The operations in this group allows you to compare a string with another lexicographically.

stringCompare()

Compares the string with the string argument. The value returned indicates the result of the lexicographical comparison:

Return value	Significance
>0	string > argument
0	string == argument
<0	string < argument

Syntax

```
long stringCompare(in string string)
```

Exceptions

```
None
```

stringNCompare()

Compares the string with the string argument for the specified number of bytes. The value returned indicates the result of the lexicographical comparison:

Return value	Significance
>0	string > argument
0	string == argument
<0	string < argument

Syntax

```
long stringNCompare(in string string, in long length)
```

Exceptions

```
None
```

Index operations

The operations in this group allow you to find the position of a substring or character, determine whether a substring or character is contained in the string or whether the string ends or begins with a substring.

indexOf()

Returns the index of the first occurrence of the substring if found; otherwise -1.

Syntax

```
long indexOf(in string substr)
```

Exceptions

```
None
```

lastIndexOf()

Returns the index of the last occurrence of the substring, if found; otherwise -1.

Syntax

```
in long lastIndexOf(in string substr)
```

Exceptions

```
None
```

indexOf()

Returns the index of the first occurrence of the character, if found; otherwise -1.

Syntax

```
long indexOf(in char token)
```

Exceptions

```
None
```

lastIndexOf()

Returns the index of the last occurrence of the character, if found; otherwise -1.

Syntax

```
long lastIndexOf(in char token)
```

Exceptions

```
None
```

stringContains()

Indicates whether the string argument is contained in the string.

Syntax

```
boolean stringContains(in string string)
```

Exceptions

```
None
```

stringContains()

Indicates whether the char argument is contained in the string.

```
boolean stringContains(in char token)
```

Exceptions

```
None
```

beginsWith()

Indicates whether the string begins with the string argument.

Syntax

```
boolean beginsWith(in string substr)
```

Exceptions

```
None
```

endsWith()

Indicates whether the string ends with the string argument.

Syntax

```
boolean endsWith(in string substr)
```

Exceptions

```
None
```

Type conversion operations

The operations in this group allow you to construct a non-string type from a string.

Some of the conversion operations accept an argument of type NumericBase. NumericBase is an enum defined in the string type.

```
enum NumericBase
{
   BaseDerived = 0,
   Base8 = 8,
   Base10 = 10,
   Base16 = 16
};
```

Use the following values to specify the corresponding base:

- `SWString::BaseDerived`
- `SWString::Base8`
- `SWString::Base10`
- `SWString::Base16`

The default is `SWString::BaseDerived` , which causes the function to determine the base from the string value:

- decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits
- octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only
- hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

stringToChar()

Returns the string value converted to a char.

Syntax

```
char stringToChar(void)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToBoolean()

Returns the string value converted to a boolean.

Syntax

```
boolean stringToBoolean(void)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToOctet()

Returns the string value converted to a octet.

Syntax

```
octet stringToOctet(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToShort()

Returns the string value converted to a short.

Syntax

```
short stringToShort(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToUShort()

Returns the string value converted to a unsigned short.

Syntax

```
unsigned short stringToUShort(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToLong()

Returns the string value converted to a long.

```
long stringToLong(in NumericBase base)
```

Exceptions

```
DSEBuiltin::ExceptionDataError
```

stringToULong()

Returns the string value converted to a unsigned long.

```
unsigned long stringToULong(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToLongLong()

Returns the string value converted to a long long.

Syntax

```
long long stringToLongLong(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToULongLong()

Returns the string value converted to a unsigned long long.

Syntax

```
unsigned long long stringToULongLong(
   in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToSize()

Returns the string value converted to a SW_SIZE.

Syntax

```
SW_SIZE stringToSize(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToFloat()

Returns the string value converted to a float.

Syntax

```
float stringToFloat(void)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

stringToDouble()

Returns the string value converted to a double.

Syntax

```
double stringToDouble(void)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

String manipulation operations

The operations in this group allow you to manipulate the contents of a string.

replaceChar()

Replaces occurrences of character `oldChar` with `newChar`

Syntax

```
void replaceChar(in char oldChar, in char newChar)
```

Exceptions

```
None
```

replaceCharAt()

Replaces character at index `pos` with `newChar` .

Syntax

```
void replaceCharAt(in long pos, in char newChar)
```

Exceptions

```
ExceptionArrayBounds
```

replaceString()

Replaces occurrences of `oldString` with `newString` .

Syntax

```
void replaceString(in string oldString, in string newString)
```

Exceptions

```
None
```

replaceStringAt()

Replaces the substring from `start` to `end` positions with `newString` .

Syntax

```
void replaceStringAt(
   in long start,
   in long end,
   in string newString)
```

Exceptions

```
ExceptionArrayBounds
```

insertChar()

Inserts the character `data` at the index `pos` .

Syntax

```
void insertChar(in long pos, in char data)
```

Exceptions

```
ExceptionArrayBounds
```

insertString()

Inserts the string `data` at the index `pos` .

Syntax

```
void insertString(in long pos, in string data)
```

Exceptions

```
ExceptionArrayBounds
```

substring()

Returns the substring from position `start` to position `end` .

Syntax

```
string substring(in long start, in long end)
```

Exceptions

```
ExceptionArrayBounds
```

trim()

Removes any leading and trailing whitespace from the string.

Syntax

```
void trim(void)
```

Exceptions

```
None
```

remove()

Removes the substring starting at position `start` and ending at position `end` from the string.

Syntax

```
void remove(in long start, in long end)
```

Exceptions

```
ExceptionArrayBounds
```

pad()

Appends a number of `data` characters to the string so that the total number of characters is `totalLength` .

Syntax

```
void pad(in long totalLength, in char data)
```

Exceptions

```
ExceptionArrayBounds (totalLength<0)
```

Case conversion operations

toupper()

Returns a string constructed by converting all characters to uppercase.

Syntax

```
string toupper(void)
```

Exceptions

```
None
```

tolower()

Returns a string constructed by converting all characters to lowercase.

Syntax

```
string tolower(void)
```

Exceptions

```
None
```

Diagnostic and initialization operations

The operations in this group allow you to valid, initialize, and set the size of strings.

stringValid()

Indicates whether the string is valid.

Syntax

```
boolean stringValid(in long maxLength)
```

Exceptions

```
None
```

reset()

Initializes the string.

Syntax

```
void reset(void)
```

Exceptions

```
None
```

resetSize()

Preallocates a buffer of the size `maxStringSize` .

Syntax

```
void resetSize(in long maxStringSize)
```

Exceptions

```
None
```

C-type string operations

The operations in this group allow you to obtain a C-type string from a string or append or copy a C-type string to a string.

getCString()

Returns a pointer to a null terminated C-type string.

Syntax

```
char *getCString()
```

Exceptions

```
None
```

attachConstBuffer()

Attach to a buffer which cannot be modified (typically shared memory).

Syntax

```
void attachConstBuffer(in char *buffer, in long length)
```

Exceptions

```
None
```

stringNCopy()

Copies a C-type string into a string object for the specified number of bytes

Syntax

```
void stringNCopy(SW_SIZE length, char *cString)
```

Exceptions

```
None
```

stringNAppend()

Appends a C-type string to a string.

Syntax

```
void stringNAppend(SW_SIZE length, char *cString)
```

Exceptions

```
None
```

14

Design Center Server Administration

This chapter describes how to set up Design Center users and control the Design Center server.

The Design Center is a Kabira Infrastructure Server (KIS) application, running on a dedicated KIS node. Each user has his or her own Design Center. Models are loaded into the Design Center using the `swbuild` command-line tool.

When multiple developers work together on a project, they can collaborate by exchanging components and importing them into their Design Centers using component specifications.

This chapter contains the following sections:

User Environment

Each Design Center server user must have the following two environment variables set, typically in their shell profile:

SW_HOME	This must be the directory where KIS was installed.
PATH	The user’s path should include $SW_HOME/distrib/platform/scripts and $SW_HOME/distrib/platform/bin, where platform is sol for Solaris or linux for Linux.

Controlling the Design Center server

The Design Center server is controlled using the swdc command line tool. This tool has the following options:

```
swdc -c command [-d] [-H path] [-m shmem size] [-n port]
    [-P port] [-p path]
        -c       configure clobber start stop forcestop status
        -d       Enable debug messages
        -H       Set SW_HOME

        Configuration Options (only valid with configure command)
        -m      Set shared memory segment size in Mb (default 200)
        -P      Set Coordinator port (randomly generated by
                default)
        -p      Set Design Center path (default )
        -w      Set Coordinator password (default none)
```

To configure and start the Design Center server:

1. go to the directory where the Design Center is to run, or specify that directory with the `-p` argument.

2. type `swdc -c start` to start the Design Center server.

This starts the Design Center.

Do not run the Design Center in a directory in the KIS installation. Instead, use a directory that is not under $SW_HOME. Make sure to run the Design Center on the same machine where KIS is installed—don't run it in a remotely mounted file system!

The command `swdc -c configure` is equivalent to `swdc -c start` , except that it stops the Design Center following installation and configuration.

Typically, a user leaves the Design Center server running. To stop the Design Center server the `swdc -c stop` command is available.

Note that by default, the Design Center uses randomly chosen free ports for the System Coordinator. You may provide specific a port number via the `-P` flag if you prefer. The port number you specify must not be in use by any other server.

You may query the state and port numbers for a running Design Center using the `swdc -c status` command.

The swdc command must know where the DC is running in order to execute commands other than configure or start. Always run the swdc command in the same directory you started the Design Center, or specify that directory with the -p argument!

The `-m` flag can be used to increase the shared memory size when compiling very large models.

Removing Design Center server

To remove the files created by the Design Center server use `swdc -c clobber.` The Design Center server must be stopped before you can remove it's files.

There is no warning before deleting the Design Center server files. Don't use the clobber command unless you really want to delete everything in the Design Center!

Index

E

F

I

K

L

M

N

O

P

R

S

T

U

V

W